Design And Analysis Of Special Purpose Multiprocessor Architecture For Discrete Event Logic Simulation.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9406027

Design and analysis of special purpose multiprocessor architecture for discrete event logic simulation

Neogi, Raja, Ph.D.

Lehigh University, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Design and Analysis of
Special Purpose Multiprocessor Architecture for Discrete Event Logic Simulation

by

Raja Neogi

A Dissertation
Presented to the Graduate School
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Computer Science

Lehigh University
Bethlehem, Pennsylvania
July, 1993

Certificate of Approval

Accepted JULY 15, 1993 (date)

Special Committee directing
the work of Raja Neogi.

Prof. Meghanad D. Wagh
Chairman of the Committee

Dr. Kalyan Mondal
Distinguished Member of Technical Staff
AT & T Bell Laboratories

Prof. Richard Decker

Prof. Madalene Spezialetti

Dedicated to

the wind beneath my wings,

Ma, Baba and wife Ajanta

Acknowledgments

I like to express my sincere thanks to my advisor, Prof. Meghanad Wagh, for his interest, advice and support throughout this research. It is hard to put down in words the value of the countless discussions through which he provided me with guidance and encouragement throughout this work. I would like to also thank other members of my Ph.D. committee — Dr. Kalyan Mondal, Prof. Richard Decker and Prof. Madalene Spezialetti, for their helpful comments and suggestions.

Above all, I owe profuse thanks to my parents, Mr. K. M. Neogi and Mrs. Krishna Neogi, without whose constant inspiration, encouragement and unfailing support, none of my work would have been possible. Also, I would like to thank my wife Ajanta for her patience and understanding during this project.

Contents

List of Figures

Abstract

With expanding design horizons, efficient simulation is an essential part of the product design and verification process. However, such simulations are extremely time consuming leading to longer product development cycles and demand use of parallel processing technology.

In this work, we propose a parallel discrete event logic simulation algorithm based on distributed event management. We analyze concurrency and correctness of the algorithm using Petri net theory. A special purpose multiprocessor architecture is designed to implement the algorithm. We present powerful integrated techniques that utilize both data and functional parallelism to improve simulation performance.

Data-parallelism is realized by statically partitioning the computational load (abstracted as cones of combinational logic) and mapping them onto processor clusters. We present a new mapping algorithm that minimizes the dynamic deviation of computation load and associated cost of global communication due to synchronization for each active simulation tick. Architecturally, such global synchronization is performrd by a two dimensional mesh network.

Each cluster is a tree of heterogeneous processing elements. The non-root elements concurrently process the computation task and cooperate with the parent processing element for local synchronization. This allows the parent processing element to look-ahead and schedule activities for the next simulation tick. We propose a new funneled-pipelining strategy that realizes functional parallelism. Our experiments have shown that most discrete event simulations have a high rate of event death along the evaluation pipe. Conventional pipelining techniques implementing such computations result in severe under-utilization of processing resource. Our pipelining strategy takes into account the funneling of the computational data space and maximizes the net evaluation throughput. The potential implications of this methodology for machine architecture are also discussed.

Finally, we present analytical techniques that measure the performance of the proposed architecture. GSPN models have been used to represent the run-time dynamics of the system. Performance Analysis based on such Markovian methods exploit the symmetries of the model to reduce the cost of numerical solution. Our workload model

incorporates the performance parameters for pipelining as well as the communication cost associated with signal synchronization.

Chapter 1

Introduction

With increased complexity of VLSI-circuits, existing Computer-Aided Design (CAD) algorithms will not be able to handle large circuits in a reasonable amount of time. It is possible to propose better heuristics to speedup CAD applications running on a uniprocessor, but the speedup obtained is going to be only marginal compared to the significantly higher speedup possible on multiprocessors. Due to time limitation on the uniprocessor, many CAD algorithms may sacrifice quality to save on time. Due to the tremendous computing power available on multiprocessors, it may be possible to get better quality solutions for the same amount of time as that spent on the uniprocessor. In the recent years, parallel processing has gained popularity due to the availability of high level languages and primitives to specify parallelism, concurrent debugging tools and better user interfaces. Parallel processing hardware has also become more affordable and accessible. Thus, parallel processing is a very attractive and cost effective alternative for speeding up VLSI CAD algorithms algorithms [1,2].

In many cases, modifying a CAD algorithm may be very simple since many CAD problems are inherently parallel. For example, many CAD algorithms are based

on a divide-and-conquer approach where smaller subproblems are first solved and then merged together to get the final solution. In such cases, the serial program can be modified to run in parallel by recognizing independent subproblems and by introducing parallel processing primitives so that the subproblems can be solved in parallel. But there can also be instances where during the process of optimization of the serial program, sequential dependencies between different subproblems may get artificially introduced. In such cases analyzing the serial program may expose very little parallelism, if any. Thus, the best serial algorithm may not be the best parallel one. The problem may have to be looked at from a different perspective, and the algorithm may have to be drastically modified to expose the parallelism inherent in the CAD problem. Discrete Event Simulation is an example of such a task. The focus of the remainder of this thesis is going to be on obtaining parallel processing solutions for discrete event simulation.

VLSI Simulation has become indispensable in the process of designing, verifying and testing complex digital systems. It is used at different stages in the design process and at different levels of abstraction. At the circuit level it is used to determine the actual electrical behavior of the implementation. Functional and register-transfer level simulation answer architectural and algorithmic questions. Gate-level simulation on the other hand, provides a relatively fast and inexpensive method of verifying intricate design detail. Over the past few years, digital designs have grown tremendously in size and complexity. There is an increasing trend towards realizing systems on a single chip with / without embedded software core(s). Such designs (often called co-designs) are better managed by dividing the verification task into gate level functional verification and gate-level multiple-delay simulation. The goal of the former

approach is to detect functionality errors while the latter exposes timing related design hazards. In either case, one has to use accelerated simulation technology to reduce the overall turnaround time significantly. Acceleration can be achieved by running conventional simulation solutions on high-performance general purpose machines or designing special purpose hardware engines to perform the same task. The first approach is definitely more flexible and a low cost way of reducing simulation time. However, it has an interesting limitation. The size of the data segment available in the primary memory is limited. For large designs, the excessive amount of roll-in and roll-out of data slows the simulation significantly. So even though the processing engine is sophisticated and fast, the processor has to idle out for the data it requires. The problem is even more acute in general purpose machines that have several hierarchical layers of cache memory. Executing cache coherence protocols [3] can steal valuable cpu cycles. In special purpose multiprocessor based accelerators, the entire data is available in the distributed memory. Each evaluation engine has its own local memory that contains a portion of the netlist. Evaluation does not have to wait for data to become available. So, even though special purpose engines are more expensive, it is possible to achieve the threshold of billion gate evaluations per second, that many designers demand. The major contribution of this work is in investigating several of the current questions that exist concerning the effective use of **parallel processing** to improve the performance of gate level system simulation. Two questions that are particularly troublesome are how one modifies the simulation algorithm and partitions it to extract maximum parallelism and how one partitions the simulation workload across the processors. Some work is reported on the first problem, but very little work has been done to address the second issue. Innovative

techniques to address both of these problems are presented and evaluated, with the intent of bringing effective gate level system simulation (million devices or more on a single chip) closer to reality.

This thesis can be divided into four parts. In chapter 2 we categorize various parallel processing techniques for VLSI simulation with their individual merits. Important issues in multiprocessor design with advantages / disadvantages of each individual approach is also examined. We survey previous work done in the area of hardware acceleration for VLSI CAD. The input data partitioning strategy for the proposed multiprocessor based simulation solution will also be discussed in this chapter.

In chapter 3 we propose a novel assignment algorithm for mapping input data graph onto processor graph. Such assignment is essential for exploiting data-parallelism in multiprocessor based solutions. It will be shown that this heuristic algorithm is efficient (*polynomial* complexity) and yields assignments with superior runtime characteristic in a simulation scenario, when compared to other known algorithms in the area. We also show that the algorithm is tolerant to errors in input parameter settings and performs equally well on all popular multiprocessor architectures.

In chapter 4 we propose a parallel simulation algorithm. The correctness of the proposed strategy will be examined. The algorithm is unique in that it exploits both domain-specific functional parallelism as well as data-parallelism. An architecture realizing this algorithm will also be proposed. In this chapter we also use a relatively new technique, stochastic petri nets, to predict the performance of our proposed architecture. Stochastic petri net theory is quickly proving to be a very powerful tool for performance prediction and analysis, particularly in situations where it would be extremely difficult to find a deterministic answer. Important design decisions will be

made based on this analysis.

In chapter 5 we propose an interconnection structure to handle asynchronous communication due to signal synchronization. Architectural details for supervising the proper sequencing of events will also be considered. Again, we use statistical techniques based on Markovian theory to analyze the performance of our simulation system. The performance analysis measure will be used to optimize architectural overhead. Finally an intra-cluster load balancing algorithm will be introduced to equitably distribute computation workload among processors.

In chapter 6 we present concluding remarks along with suggestions for future research.

Chapter 2

Motivation and the Model

2.1 Introduction

This chapter examines the various issues and options that one has to consider to develop a multiprocessor based logic simulation solution. The discussion is abstracted from actual implementation of algorithms so as to explore the parallelism inherent in the problem rather than trying to parallelize a particular serial algorithm. Previous research in the development of hardware accelerators for simulation is also reported here. In order to exploit parallelism, one has to not only consider the level of simulation but also the abstract netlist view and the granularity of simulated time. The last section examines these issues in depth.

2.2 Logic Simulation

The simulation of VLSI systems is performed at different levels of abstraction, ranging from architectural to gate and switch level to circuit simulation [4]. In the case of high level simulation one abstracts largely from the actual implementation and mod-

els the system as an interconnection of functional blocks specified as subroutines in a high level programming language. In gate level and switch level simulation, circuits are modeled as the interconnection of boolean gates and idealized transistors respectively, while circuit simulators are based on more detailed physical device models. Corresponding to the level of abstraction, the time granularity in the different types of simulation varies: architectural simulation typically has a time resolution based on complete bus or machine cycles; gate and switch level simulation is based on clock cycles; circuit simulators in contrast compute complete continuous waveforms. The time granularity in the simulation is an important parameter for the performance of parallel simulation schemes [5–7].

Logic simulators are in widespread use as tools to analyze the behavior of digital circuits. Logic simulators rely on abstract models of the functioning of a digital system and yield discrete value outputs [8]. For gate level simulation, circuits are described in terms of primitive logic gates. They are typically evaluated through table look up or by calling a software function. Gate level simulators are popular, because they can be implemented efficiently and because the gate level model is in direct correspondence to the boolean equation representation of digital systems. In contrast, a switch level simulator operates directly on the transistor-level description of a circuit. The transistors are modeled to behave like switches and the steady state of the transistor network is computed iteratively. With suitable adjustments to the evaluation engine of a gate level simulator, most circuits typically verified using a switch level simulator can be verified using an enhanced gate level simulator. As a result, simulation models in close correspondence to actual physical implementation, for example circuit description extracted from layout, can be simulated directly. The

```
Simulate_event_Driven ( ) {
   for all Input Vectors {
      Update Input Signals
      Schedule Connected Elements for Evaluation
      while (elements left for evaluation) {
         Evaluate Element
         If ( change on output) {
            Update Fanout
            Schedule Signal Update
         }
      }
   }
}
```

Figure 2.1: Event Driven Simulation Algorithm

discussion above is restricted to discrete event simulations. An *event* is a change in the value of a signal and an element of the model to be simulated is re-computed only if one of its inputs changes. This is in contrast to an all-event simulation (employed for example in the EVE hardware accelerator), where the entire model is updated for each time step. The main steps in event driven simulation are summarized in Figure 2.1.

2.3 Parallelism in Simulation

One can extract two types of parallelism in logic simulation viz. algorithmic or functional parallelism and data or model parallelism [9, 10]. In functional parallelism, functional structure of an algorithm is partitioned so that different functions can

be evaluated in parallel. In event driven logic simulation, event evaluation, signal scheduling and device scheduling can be performed in parallel. The number of such functions that can be evaluated in parallel is limited and thus the method does not scale well with the number of processors. If data elements are partitioned so that they can be processed in parallel, it is called data partitioning. In event driven simulation, events in a circuit propagate concurrently along many different paths. Consequently, different elements are active at the same time and can be computed by several processors in parallel. The improvement in computation time by complexity reduction has a penalty. The partitioned devices residing in different processors need to synchronize, before the simulation tick can be advanced. Therefore, a partitioning strategy has to be designed to minimize communication overhead and maximize concurrency. Data parallelism scales very well with the number of processors. The parallelization methods proposed in this work are based on both data and functional parallelism.

2.4 Synchronization Approaches

2.4.1 Event Synchronization

The task of synchronizing a number of cooperating processes has been studied extensively in the literature [11–13]. As discussed above, if data parallelism is used, a partitioned computation node needs to communicate its state change to its temporal neighbors. Such neighboring nodes may be located on different processors and therefore the update process involves communication latency. This type of synchronization between the sending and receiving nodes is known as signal synchronization or event synchronization. Most software discrete event simulators work on a single global

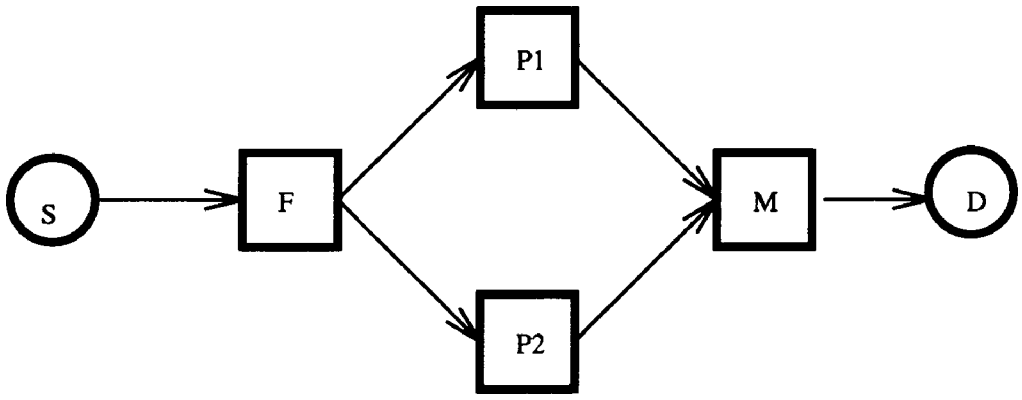
event list. This synchronization approach can be easily adopted in shared memory multiprocessor systems. Each computing node communicates its state change to its neighbors through the global event list. Both the high bandwidth of communication and the resultant communication latency limits the usefulness of this approach. Another alternative is that each individual processor maintain its native event list. Only non-native events need travel through the event or signal synchronization plane. This approach is particularly suitable for distributed simulation or MIMD machine implementations. The communication latency can be controlled by suitably selecting an interconnect structure, rich in physical connections between processors.

2.4.2 Time Synchronization

Another major issue in the development of distributed simulation algorithms is how simulated time is synchronized between processors. There are two major classes of techniques addressing this issue, the global clock approach and the local clock approach. In the global clock approach, a single clock is commom to all processors. All the processors proceed in lock step with respect to the simulated time, which is under the control of a single master controller. Once all the processing has been completed for the current time, the master controller sends a message (a *start* message) to each processor indicating that time can be advanced to the next time point. When each processor completes its task at this new time, it sends a message (a *done* message) to the master indicating it is done. When all processors have done this, the master can repeat the above cycle. In discrete event simulation implementations that fold in variable node computation latency, events are sparsely distributed along the time axis. To avoid wasteful overhead in the form of start and done messages for idle ticks,

one can use a slight variation of this algorithm. Each processor includes in its done message to the master, the future time point when it has more work to do. The master now has sufficient information to be able to skip time points that have no work to be done, and can include next time information in its start message to each processor.

A second time synchronization approach is to allow each processor to maintain its own local time. Different processors are allowed to be ahead of or behind one another in simulated time, but must be synchronized so that the correct sequence of events still take place. There is no global time-keeper in this approach. In the Chandy Mishra Algorithm (CMA) [14,15], time stamp of an event forms an important component of message transaction between processors. The local time of each processor is always the minimum of the time-stamps received along all input channels. The time-stamp of a message sent on an output channel must always be greater than or equal to the local clock value. Assuming, messages along a channel are received in monotonically increasing order only, the causality constraint is assured. The CMA algorithm is a conservative approach for advancing simulated time. One major bottleneck of this approach is the deadlock detection and recovery. A deadlock is a state of the system when none of the processors can advance its local clock, without external intervention. Figure 2.2 below illustrates one such problem. Assume that messages from S to F are routed to P1 and no messages are routed to P2. Messages arriving at P1 cause messages to be sent on the output channel of P1 and arrive at M. However, M will never receive a message from P2. Now M is blocked and its local clock can never be advanced and no messages can be sent to D. When P1 completes processing all messages it received from S the simulation clock halts or deadlocks. The extra amount



[S] Source Process [F] Fork Process
[M] Merge Process [D] Destination Process
[P1, P2] Intermediate Process

Figure 2.2: A Simulation that cannot progress

of parallelism gained by relaxing faster processors having to wait for slower ones, is negated by the extra amount of work required to detect a deadlock and recover from it. Many fast deadlock detection, recovery and avoidance techniques have been proposed in the literature, but none of these approaches show promise of being adapted in an accelerated simulation environment.

2.5 Previous Work

Several research projects [9] have investigated the use of dedicated hardware accelerators for VLSI simulation. The Yorktown Simulation Engine (YSE) [16] and its successor EVE [17] performs compiled code simulation. That is, rather than evaluating components when one of their inputs change state, every component is evalu-

Task	% Instructions
Queue Operations	41
Functional Evaluation	35
Netlist Operations	20
Other Overheads	4

Figure 2.3: Logic Simulation Task profile

ated at each point in simulated time. This has the disadvantage of requiring more component evaluations to take place, but eliminates the requirement of maintaining an event queue to determine when and where circuit activity takes place. The Zycad Logic Evaluator [18] uses an event-driven global clock to synchronize events on special purpose processors that are interconnected via a high speed bus. Another popular hardware accelerator, MARS [19, 20], uses a pipelined design where each functional unit performs a specific subtask. The hardware consists of processing elements connected to a crossbar switch. The processors are microprogrammed engines with special message passing and table manipulation capabilities. The simulation algorithm is partitioned among the processors. Coleman and Ambler [21] profiled a logic simulator written in a high level language. Results obtained by counting the number of machine instructions executed are shown in the Table 2.1 above. This profile highlights the necessity to accelerate queue manipulation, functional evaluation and netlist operations to avoid a large efficiency degradation. A clear suggestion is that queue manipulation and functional evaluation should be handled by dedicated hardware and netlist operations eliminated by compiling a separate code sequence for each element.

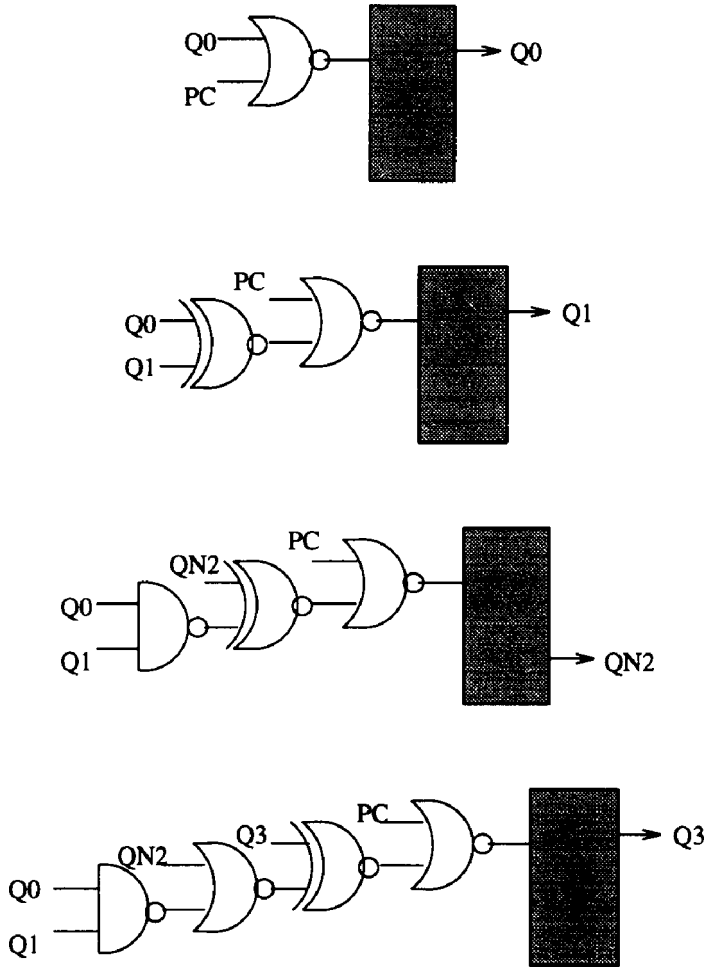
2.6 Input Data Model

In our proposed approach, all combinational devices are assumed to have zero-delay and all sequential elements unit-delay. Sequential devices, alternatively referred to as blocking devices in this research, do not necessarily have to be latches or flip-flops. Functional blocks like memory blocks and custom blocks (gate level adaptation of switch level descriptions) could also be defined as blocking devices. The flat gate level netlist description is preprocessed to color blocking devices and extract cones [22] of combinational logic on input ports of blocking devices. Some of these blocking device ports have control property, some data. The data ports could either be enabled by the cumulative assertion of control ports as in a horn clause, or groups of control port assertion could enable specific data ports. Computing the steady state of the circuit reduces to first evaluating zero-delay combinational logic cones in one pass and updating all blocking devices for the next clock phase. In all real life synchronous designs, most logic cones are free from asynchronous loops. However, the zero-delay view of logic cones can introduce some artificial loops. Such loops can be broken by suitably injecting fictitious delay element (also called pseudo blocking devices with delay) in the loop. The presence of such delay element should not alter the logic. Primary outputs, that do not belong to any blocking devices are also labeled as pseudo-blocking devices (without delay). Pseudo blocking devices are characterized by the presence of universally enabled control condition. The data-flow across such blocking devices is not guarded. This model allows levelizing logic cones. Levelized logic, free from loops can be computed very fast with specialized hardware. Thus one can consider the logic cone as a form of super-gate that gets excited only when atleast one of its input changes. The model has an interesting lookahead property. Deep,

wide and therefore expensive data cone computation can be masked by pre-evaluating shallow control cones. Encapsulation of combinational logic also eliminates fictitious glitches. This way, the model captures the steady state behavior of the machine at all points of simulated time. Mathematically, the model can be represented as a condensed di-graph. Blocking devices with their cones represent the supernodes and inter-cone data-flow paths represent the directed edges. Supernodes have weights that represent computation size. Edges have weights too, that quantize the amount of data-flow. Figure 2.3 shows an example of a digital circuit and its blocking devices with logic cones, and Figure 2.4 depicts the equivalent condensed digraph.

2.7 Summary

In this chapter, background material for VLSI simulation was discussed. The discussion touched base on various ways of exploiting parallelism in simulation. Synchronization is a major issue in any multiprocessor design. Different synchronization methods were discussed with advantages / disadvantages in each case. Previous work in the field of *hardware acceleration for CAD* was examined with specific emphasis on simulation. Finally, input data model for the proposed multiprocessor based simulation solution was presented.



 # connections not shown

Figure 2.4: Example of a four bit counter

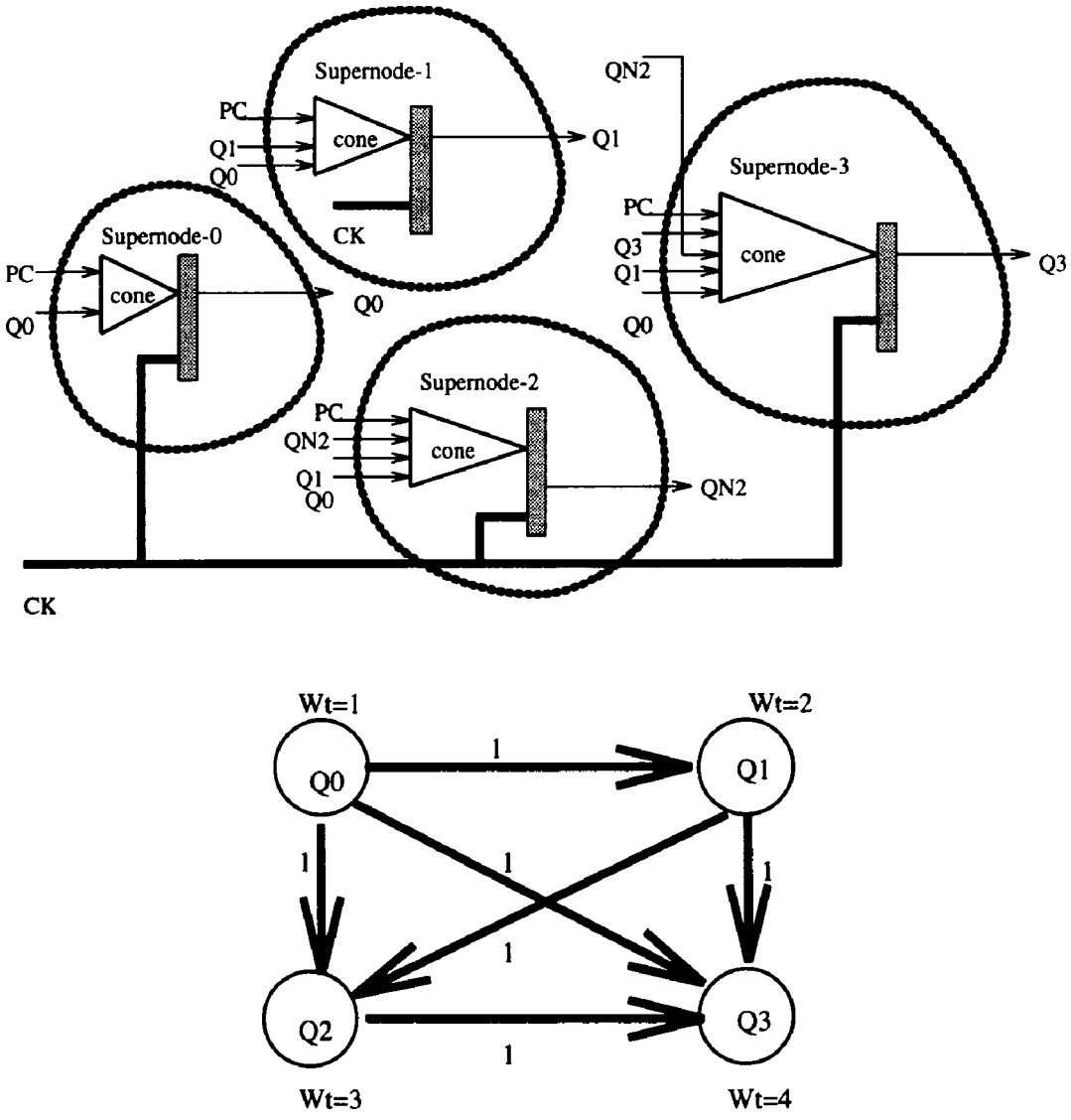


Figure 2.5: Equivalent Condensed Digraph

Chapter 3

Algorithms for Implementing Data-Parallelism

3.1 Introduction

Irregular task graphs arise in many simulation environments. Both Partitioning of such graphs and its subsequent Mapping on regular parallel architectures is an NP hard problem and has attracted wide attention. A heuristic method for partitioning the task graph is presented in section 3.2. The method utilizes domain specific characteristics to yield a condensed di-graph, ideal for coarse grain parallel simulation. The next three sections (3.3-3.5) address the problem of mapping the input graph onto a multiprocessor. Mapping or assignment problems are particularly difficult because one has to address two conflicting requirements. Computational load across processing elements (PE) need to be well balanced. Efforts to balance the computation load may result in mapping that has extremely high communication cost. This is because, PEs that need to exchange data frequently may not be neighbors in the PE interconnection graph. In case of regular paradigms such as parallel divide-and-conquer, it

is many times possible to properly map the task graph onto the host (architecture) graph such that load imbalance and communication costs are minimized [23, 24]. In case of simulation algorithms, the complexity and interconnection of task graph depends on the particular simulation scenario and cannot be predetermined or expressed through an algebraic formula. Previous efforts or logical extensions of previous work to map input task graph onto processor graph is discussed. The Major contribution of this section is the mapping algorithm based on minimizing the maximum penalty due to improper assignment at every iteration [25]. Results of such mapping strategy on popular architectures indicate that it yields superior runtime characteristic compared to the other known algorithms.

3.2 Partitioning Algorithm

For any parallel processing application, one has to partition the input data into blocks of computation load that can be mapped onto the multiprocessor system. It is highly desirable to have blocks of nearly equal computational complexity. Also, blocks need to be identified in such a way that the number of physical links connecting the blocks is as small as possible. Flat VLSI circuit descriptions result in graphs that are highly irregular and random. If such a graph is directly mapped onto a massively parallel multiprocessor system, the load deviation may be optimized but the communication complexity becomes unmanageable. One should note that massively parallel simulation is only feasible if the simulated time is synchronized in a distributed manner. The alternative is to encapsulate chunks of logic into blocks that closely meet the criteria mentioned above. Such encapsulation could be random or based on a set of heuristics. In this work, blocks are formed by first identifying the block

master, which happens to be the sequential(blocking) device. Cones of combinational logic that feed the inputs of a blocking device are identified as the slaves of the block master. This approach limits the physical interconnection between blocks, because, in a typical digital design the sequential element on an average fans out to fewer devices. The computational complexity of individual blocks do vary, but the deviation is fairly well bounded. Control cones are shallow and add little to the complexity. The depth of data cones is limited by the operating frequency of a chip. Since, latch to latch timing is critical for high-performance VLSI circuit designs, this cone based heuristic results in fairly optimal partitions. The parallelism grain size is also transformed from fine to coarse. In a typical digital design, sequential devices constitute only 10-15% of the device count. Coarse grain parallelism is a must for simulation solutions that manage simulated time globally. Figure 3.1 illustrates the *Cone Extraction Algorithm.* Individual blocking devices are first colored. The algorithm recursively traces back along each input of the blocking device until it hits a primary input or another blocking device. This defines the boundary of the logic cone. Each input of a blocking device has a cone that can be labeled as control cone or data cone that is characterized by consulting the cell-type attribute of the blocking device data base. Control cones are shallow and far more active. Data cones on the other hand are wide, deep and much less active. Data cone computation is guarded by grouped control cone assertions. Each blocking device (or block master) with its bag of logic cones constitutes a supernode. Data-flow between individual blocks is modeled as edges in the condensed input di-graph. The next section examines the problem of mapping the condensed input di-graph onto the processor graph.

```
Output Cones () {
    for every Colored-Cell, 1<=c<=N
        for every Fanin f, 1<=f<= |Inputs|
            TraceBack (Fanin)
}

TraceBack (Fanin) {
    if (Fanin is a Blocking Device)
        Print Id for Blocking Device
    else
        for every Fanin f, 1<=f<=|Inputs|
            Traceback (Fanin)
}
```

Figure 3.1: Cone Extraction Algorithm

3.3 Assignment Problem Description

On an abstract level, the mapping problem has two inputs. The first is the task graph $G_d = (V_d,\ E_d)$ whose vertices V_d represent various tasks to be done and whose directed edges E_d indicate data or information flow between these tasks. Each task is assigned a weight (*size of the node*) which is representative of the computation associated with that node. Similarly, each edge has a weight that corresponds to the amount of communication between the pair of vertices connected by the edge.

The second input to a mapping problem is the processor graph, $G_p = (V_p,\ E_p)$, describing the MIMD multiprocessor system to be used. Vertices V_p of this graph represent the processors and edges E_p represent the interconnection between these processors.

The mapping problem can be modeled as a function $\phi : V_d \rightarrow V_p$ defined as $\phi(d_0) = q$ if task d_0 is assigned to processor q. In a realistic situation of simulation of large task graphs, it is generally assumed that the number of processors is much smaller than the number of tasks. Thus the function ϕ is a many to one function. During each simulation clock tick, one has to evaluate the complete task graph and transmit the information necessary for the next tick evaluation according to the edges E_d of the task graph. The computation in each processor can take place concurrently and the communication between any pair of processors can be assumed to be concurrent for simplicity. However the simulation clock tick cannot be advanced until all the processors have completed evaluation. Thus the effectiveness of a mapping scheme may be judged by computing the largest time taken by any processor to finish the computations and communication assigned to it. The time complexity of a task partition generated by a function ϕ can be described by

$$\max_{p \in V_p} \{ \lambda \sum_{\phi(n)=p} w(n) + \sum_{\phi(n_1)=p, \phi(n_2)=q} w(n_1, n_2) D(p, q) \}, \tag{3.1}$$

where $w(n)$ denotes the weight of task $n \in V_d$, $w(n_1, n_2)$ is the weight of edge connecting nodes $n_1, n_2 \in V_d$ and $D(p, q)$ is the shortest distance between nodes p and q in V_p. The constant λ represents the cost of one unit of computation relative to one unit of communication. λ is based on the definitions of weights of vertices and edges in task graph G_d as well as on the architectural parameters such as the computational and communication speeds. In (3.1), the first sum gives the computational load on each processor and the second sum which is evaluated over all edges $(n_1, n_2) \in V_d$ provides the communication load.

A good mapping algorithm should provide a task distribution that gives the smallest possible total cost as described by (3.1). However, in a typical simulation envi-

ronment and particularly in event driven simulations, the amount of computation associated with each task $n \in V_d$ as well as the amount of information transmitted between task nodes n_1 and n_2 dynamically changes as the simulation progresses. This is because of the fact that all the tasks may not have to be evaluated at every clock. This means that the performance of a particular task partition may vary at every clock tick. The only alternative to this, dynamic partitioning which implies task migration during simulation, is even more expensive. The mapping algorithms are therefore derived using constant weights of nodes and edges of V_d. Also, cost function used employs *load deviation* rather than *load on a particular processor*. We believe that load deviation is a more stable parameter in the face of dynamically varying complexity of tasks during simulation. It is more reasonable to assume that if $|V_d| \gg |V_p|$, average load on all processors (each of which is a host to a large number of tasks) will show similar behavior from one clock to the next resulting in a stable load deviation.

3.4 Assignment Algorithms

Several techniques for task assignment in parallel and distributed environment have been reported. Bokhari has shown that the general mapping problem is NP-complete, and has proposed the "pairwise interchange" heuristic [26]. However, this method assumes that the cardinality of vertices in the task graph is identical to the cardinality of vertices in the processor graph. Sarkar and Hennessey have used a list scheduling algorithm with a cost function reflecting completion time [27]. Because of the global clock synchronization, load balancing (including the communication costs) is even more important in simulation applications. A poor mapping of G_d on G_p can easily

result into a more heavily loaded (and therefore a slower) processor which may hold up the advancing of simulation clock thus idling the rest of the processors. Since simulation is generally a very time intensive computation, such inefficiencies result into unacceptable performance. Yet, little progress in task assignment is reported in simulation literature. Kravitz and Ackland have reported two mapping strategies in the CEMU system. Their first method maps randomly ordered tasks onto processors with the smallest load one by one and the second maps pre-sorted tasks onto processors with smallest load [28]. Both of these methods try to obtain an even load distribution, but no effort is made to optimize communication cost. Agrawal has considered a partitioning technique in which all strings of vertices are lumped onto the same processor [29].

The task assignment problem in the context of a simulation algorithm is similar to the general task assignment problem. But in the general task assignment, the edges of the task graphs represent both data and temporal dependency. The computational model of a simulation algorithm allows one to assume the representation of data dependency only. In every time step of the simulation, the activity distribution is input stimuli dependent and so temporal dependency is not built into the task graph. Thus, the objective of this work is to start from a task graph G_d giving the task complexities and the data dependencies and a processor graph G_p and to obtain a many-to-one mapping ϕ from G_d to G_p to achieve better load balancing *and* minimal communication costs.

Four Algorithms are presented in this section. The comparison of their performance is presented in the next section. The first of these algorithms shown in Figure 3.2 is rather simple. It starts by sorting vertices of G_d in the decreasing order

```
Algorithm_1 (V_d, N) {
  A[0, K − 1] ← sort vertices in descending order of size
  for (i = 0; i < K; i++) {
    Find the most lightly loaded processor
    Assign A[i] to that processor
  }
}
```

Figure 3.2: The simple load balancing algorithm.

of their computation sizes. It then allocates them one at a time to the most lightly loaded processor at that stage. It thus tries to balance the loads without any regard for the resultant communication costs. The reason for presorting of the vertices is rather simple. Every time a new vertex is assigned in accordance with this algorithm, the load imbalance cannot increase by more than the size of the new vertex. Thus assigning smaller nodes later has the desirable effect of minimizing the final load imbalance.

The other three algorithms do take the communication cost into account while partitioning the graph V_d. Figure 3.3 shows the the pseudocode for the second algorithm. In this algorithm, the computational tasks are presorted in decreasing order of their computational load. As before, this helps ensure lower overall load imbalance. In every new iteration of the algorithm the most complex task amongst those that are left over is considered for assignment.

To assign a new task, the impact of its assignment to each processor in the architecture is evaluated. This impact is measured in terms of the cumulative communication cost and the deviation of load in the system. In simulation systems both load im-

Algorithm_2 $(V_d,\ N,\ \gamma)$ {

$A[0,\ K-1] \leftarrow$ sort vertices in descending order of size

for $(i = 0;\ i < K;\ i++)$ {

$d \leftarrow A[i],\ \ L(p) \leftarrow \Sigma_{\phi(v)=p}|v|$

$tavg \leftarrow (\Sigma_p L(p) + |d|)/N$

$tvar \leftarrow \Sigma_p (L(p) - tavg)^2$

for every processor p, $(1 \le p \le N)$ {

$\mathrm{LD}(d,\ p) \leftarrow \sqrt{tvar - (|p| - tavg)^2 + (|p| + |d| - tavg)^2}$

$\mathrm{CCC}(d,\ p) \leftarrow \mathrm{C}(p) + \Sigma_{d_j \in V_d} w(d_j,\ d) \cdot D(p,\ \phi(d_j))$

$\mathrm{F}(d,\ p) \leftarrow \gamma \cdot (\mathrm{LD}(d,\ p)) + \mathrm{CCC}(d,\ p)$

}

Find p with smallest value of $\mathrm{F}(d,\ p)$

Assign d to p

}

}

Figure 3.3: The simple algorithm for minimizing load imbalance and communication.

balance and excessive communication are undesirable since they imply slowing down of the simulation. We therefore choose the objective function as γ(Load Deviation LD) + (Cumulative Communication Cost CCC) where γ is the user defined bias of load deviation over communication costs. At any iteration, we assign the largest unassigned vertex to the processor that yields the minimum value of this objective function.

To calculate $CCC(d_0, p)$, we examine all the neighbors of d_0 in the graph G_d. Let $D(p,\ q)$ denote the distance between processors p and q if both are assigned otherwise, let $D(p,\ q) = 0$. Then

$$\mathrm{CCC}(d_0, p) = \mathrm{C}(p) + \sum_{d_i} w(d_0,\ d_i) \cdot \mathrm{D}(p, \phi(d_i)),$$

where C(p) is the total amount of communication cost bourne by processor p after last assignment. The deviation of load, LD(d_0, p) is calculated by noting the change in load variance of the system. The load variance change results, because, we are examining the impact of assigning the new vertex d_0 (a computation load) onto processor p. $tavg$ indicates the new average load on processor p after assigning the vertex to p. The variance of load after last assignment is represented by $tvar$. The objective function combines the impact of both load deviation and communication cost, using bias constant γ.

In the remaining two algorithms shown in Figure 3.4 and Figure 3.5, the vertices are not presorted, rather, a merit function is defined for each vertex. The vertex for which this merit function is optimum is due for assignment for that stage of iteration. For both of these algorithms the same objective function γ(Load Deviation LD) + (Cumulative Communication Cost CCC) is used. At each iteration, the entire pool of un-assigned vertices are considered. For each such vertex a dynamic profile of its

```
Algorithm_3 (V_d, N, γ) {
  while there exists an unassigned vertex in V_d {
    for every unassigned vertex d {
      for every processor p, (1 ≤ p ≤ N) {
        LD(d, p) ← load deviation when d is assigned to p
        CCC(d, p) ← comm. cost when d is assigned to p
        F(d, p) ← γ· LD(d, p) + CCC(d, p)
      }
      Find cheapest ψ(d) processor selection for d
      Merit(d) ← F(d, ψ(d))
    }
    Find the vertex d_s with the smallest Merit
    Assign vertex d_s to processor ψ(d_s)
  }
}
```

Figure 3.4: Algorithm to choose the best vertex-processor pair to maximize correct selection rewards.

```
Algorithm_4 (V_d, N, γ) {
  while there exists an unassigned vertex in V_d {
    for every unassigned vertex d {
      for every processor p, (1 ≤ p ≤ N) {
        LD(d, p) ← load deviation when d is assigned to p
        CCC(d, p) ← comm. cost when d is assigned to p
        F(d, p) ← γ· LD(d, p) + CCC(d, p)
      }
      Find cheapest ψ(d) and second cheapest ψ'(d) processor
        selection for d
      Merit(d) ← F(d, ψ'(d)) - F(d_0, ψ(d))
    }
    Find the vertex d_s with the largest Merit
    Assign vertex d_s to processor ψ(d_s)
  }
}
```

Figure 3.5: Algorithm to choose the best vertex-processor pair to minimize wrong selection penalty.

impact on being assigned to each of the available processors is computed. A processor assignment that minimizes the objective function is the chosen processor $\psi(d)$ for that vertex d.

In the third algorithm, the merit of vertex d is defined to be the value of the objective function when d is mapped on $\psi(d)$. A vertex is assigned at a given iteration if its merit (as defined here) is minimum. This ensures that at every step, the most profitable vertex-processor pair is selected. Thus this technique tries to maximize the correct assignment rewards. In the fourth algorithm, rather than directly maximizing the correct assignment rewards, the wrong assignment penalties are minimized. This is achieved as follows. When the assignment of a vertex d to all possible processors in the architecture is considered, we determine two processors $\psi(d)$ and $\psi'(d)$ which produce the smallest (x_0)and the next higher (x_1) values of objective function respectively. The merit function that now determines if the vertex is assigned at that iteration, is defined to be the difference in x_1 and x_0. This merit function value, in some sense, provides an estimate of the penalty if the vertex d was wrongly assigned to $\psi'(d)$ rather than to $\psi(d)$. The vertex that gets chosen for assignment is the one that has the largest value of the merit function described above. Thus algorithm 4 minimizes the wrong assignment penalty.

3.5 Algorithm Evaluation and Analysis

The four algorithms presented in the previous section were applied to simulation of VLSI logic circuits. Five typical circuits derived from a wide range of dissimilar applications were used: a multiply-and-accumulate unit (macc), an arithmetic/logic unit for a controller (calu), a traffic light controller (tlc), a microprocessor interface (mintl)

Circuit	Nodes			Edges
graph	Total	Ave size	Ave out-degree	Total
macc	42	449.36	21.0	546
calu	25	2.0	1.25	5
tlc	21	12.90	8.73	131
mintl	40	6.80	4.88	88
lfsm	127	73.29	12.45	1171

Figure 3.6: The characteristics of the five chosen graphs

and a very large and complex finite state machine (lfsm). The circuits were flattened and preprocessed to abstract the the output *cones* view. Each sequential device along with its combinational logic cones represented a vertex of G_d. An edge was drawn from vertex n_1 to n_2 if any output of the sequential device n_1 is used to compute (through combinational logic cone) the excitation of n_2. Weights of vertices were chosen to be the number of combinational gates in the vertex. This weight represents the computational effort that must be spent to evaluate the vertex. Weights of all the edges was considered unity because in the present application, the information carried between any pair of vertices is the value of the logic level of the corresponding electrical signal and was therefore considered a message of the same length. The properties of the five chosen circuit task graphs are given in Table 3.1. One can see from Table 3.1 that the graphs used for testing our algorithms are highly irregular, random and fairly typical of the graphs one would deal with in VLSI simulation.

MIMD model of parallel computation was chosen for the obvious reason that each processor is to independently evaluate its assigned data region and perform the associated communication. Experiments were performed with popular interconnect

schemes such as grid (mesh with no wraparound connections), hypercube and crossbar switch. Without loss of any generality, processor cardinality of each architecture was assumed 16. This is consistent with the fact that massive number of processors will slow down the simulation clock synchronization step. Of the architectures chosen, the hypercube and the crossbar are symmetric networks, i.e., from the perspective of each processor in the architecture, the rest of the network looks the same. The grid on the other hand, is nonsymmetric because the corner and edge processors look very different from the internal ones. However, grid was the only planar network amongst the three studied and in VLSI implementation of a parallel processor (as in a hardware accelerator), planarity is a highly desirable property.

The performance comparison of the four algorithms on the chosen architectures is shown in Figures (3.7) - (3.9). These figures measure the performance of Algorithms 2, 3 and 4 relative to that of Algorithm 1 as parameter λ which characterizes complexity of unit computation with respect to unit communication is varied between 0.1 and 1.2. Note that algorithms 2, 3 and 4 require user to decide on the bias factor γ. During the partition design stage, γ should be set equal to the ratio of the costs of unit computation and unit computation. However, during simulation, the amount of computational task associated with the vertices may change because many tasks may be active only in certain clock ticks. Thus, the value of γ can only be *estimated* roughly by taking into account the definitions of the *weights of a vertex and a link* in the application and the computational and communication speed of the architecture. Algorithms 2, 3 and 4 were evaluated with $\gamma = 0.8$.

It can be seen from the figures above that the new algorithm 4 is substantially better than the other algorithms for a wide range of λ. In order to study the effect

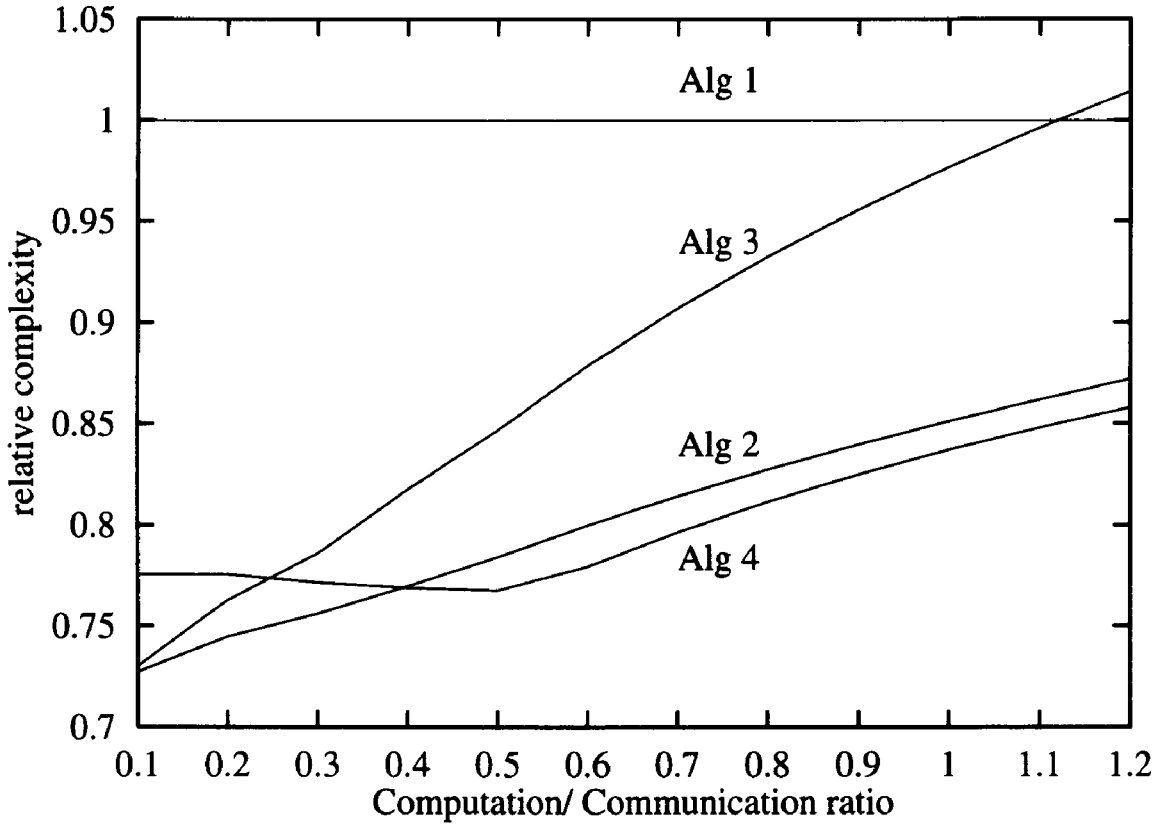


Figure 3.7: Comparison of four algorithms on a Hypercube

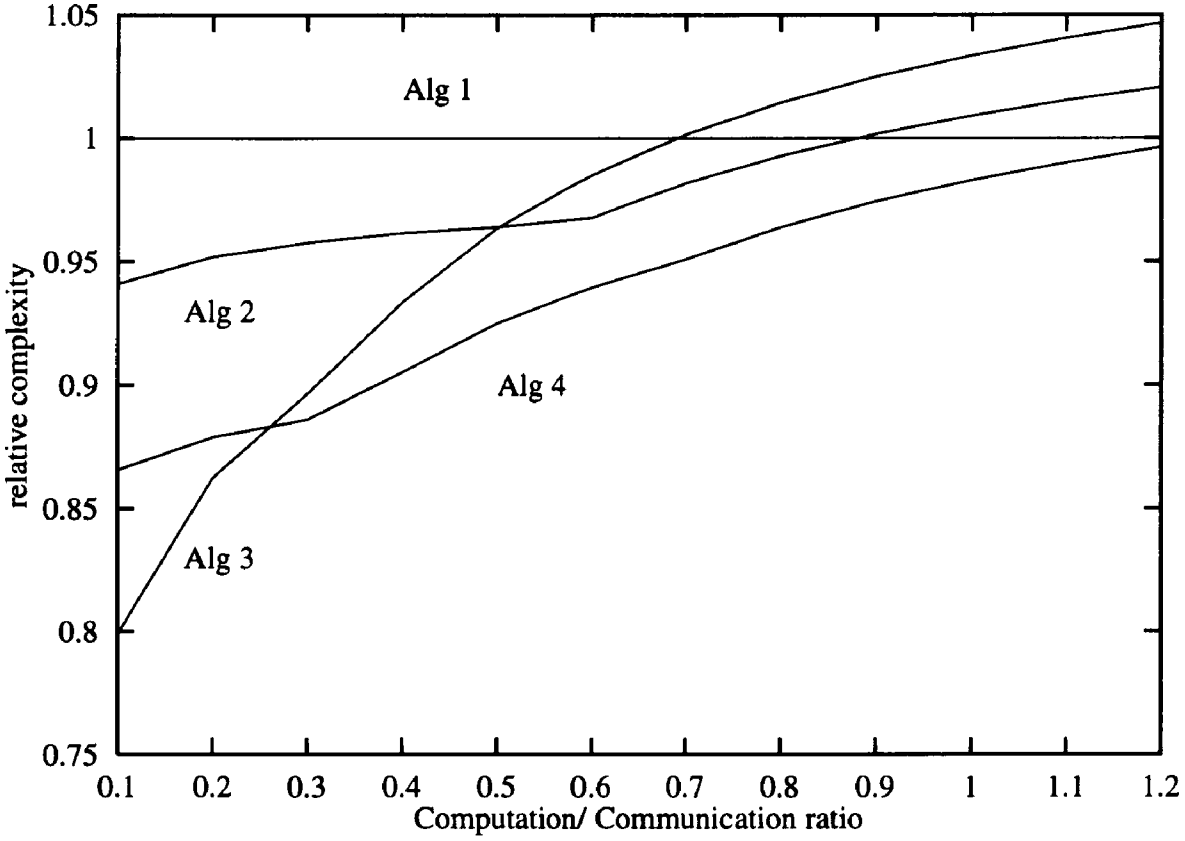


Figure 3.8: Comparison of four algorithms on a Grid

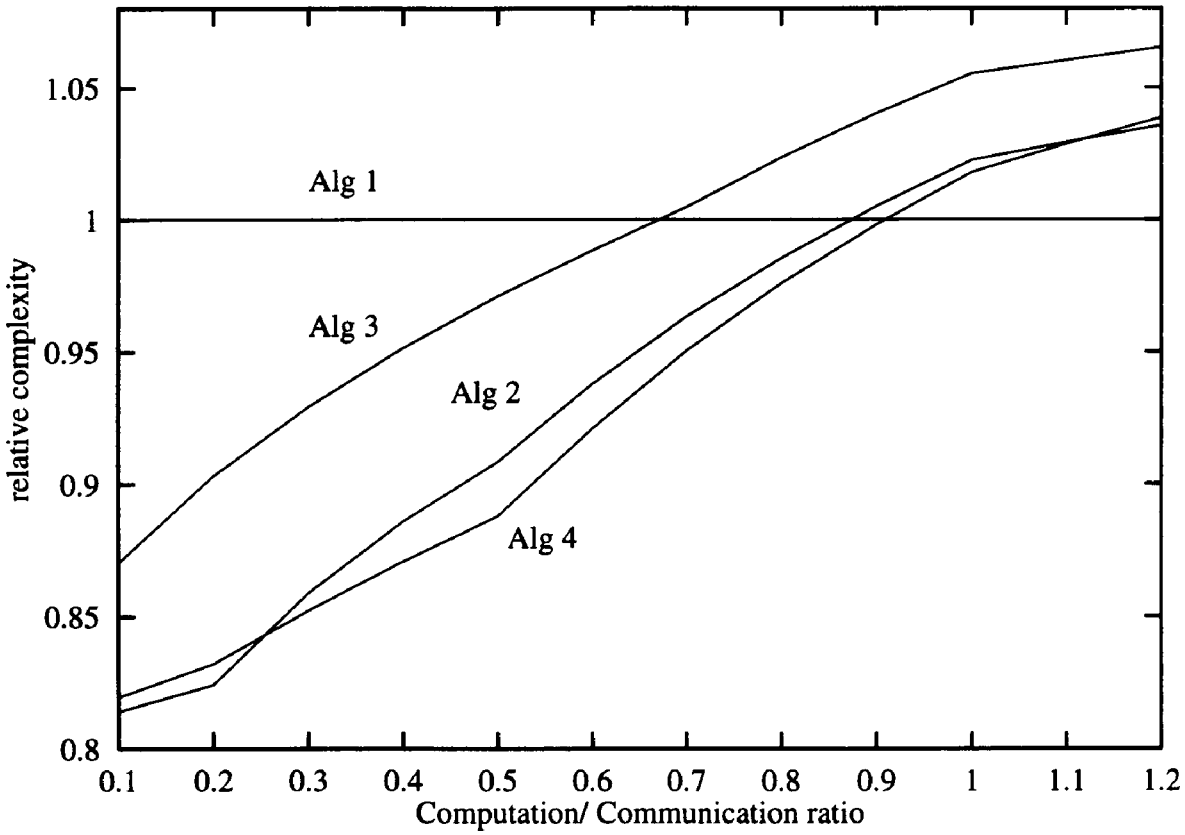


Figure 3.9: Comparison of four algorithms on a Crossbar

λ	partition complexity		
	$\gamma = 0.5$	$\gamma = 0.8$	$\gamma = 10$
0.1	260.4	264.3	280.3
0.2	405.8	414.6	430.6
0.3	551.2	564.9	580.9
0.4	698.6	715.2	731.2
0.5	851.5	865.5	881.5
0.6	1004.4	1015.8	1031.8
0.7	1157.3	1166.0	1182.1
0.8	1310.2	1316.4	1332.4
0.9	1463.0	1466.7	1482.7
1.0	1616.0	1617.0	1633.0
1.1	1768.9	1767.3	1783.3
1.2	1921.8	1917.6	1933.6

Figure 3.10: The effect of different γ values on complexity of partitions of macc graph on a degree 4 hypercube

of errors in γ prediction on the final partitions, we applied algorithm 4 with γ equal to 0.5, 0.8 and 10 to the macc graph. The performance of the resultant partitions is given in Tables 3.2-3.4.

Let K and E denote the total number of vertices and the total number of links respectively in the task graph V_d. Represent by N, the total number of processors in the multiprocessors system. In Algorithm-1, the sorting step takes $O(K \log K)$ time. Within each of the K iterations over variable i, one has to find the most lightly loaded processor, an action that would consume $O(N)$ time. Thus all the iterations over i would take $O(KN)$ time. Thus the total time complexity of Algorithm-1 is

λ	partition complexity		
	$\gamma = 0.5$	$\gamma = 0.8$	$\gamma = 10$
0.1	321.0	299.0	314.0
0.2	468.0	434.0	436.0
0.3	615.0	569.0	572.0
0.4	762.0	715.6	713.0
0.5	909.0	868.5	858.5
0.6	1056.0	1021.4	1011.4
0.7	1206.7	1175.0	1164.3
0.8	1368.8	1334.0	1317.2
0.9	1534.6	1493.0	1470.1
1.0	1701.0	1652.0	1623.0
1.1	1867.4	1811.0	1775.9
1.2	2033.8	1970.0	1928.8

Figure 3.11: The effect of different γ values on complexity of partitions of macc graph on a 4×4 grid

λ	partition complexity		
	$\gamma = 0.5$	$\gamma = 0.8$	$\gamma = 10$
0.1	204.4	204.6	206.3
0.2	353.8	350.2	356.6
0.3	503.2	495.8	506.9
0.4	652.6	644.2	657.2
0.5	802.0	796.5	807.5
0.6	951.4	949.4	957.8
0.7	1102.3	1102.3	1108.1
0.8	1255.2	1255.2	1258.4
0.9	1408.0	1408.1	1408.7
1.0	1561.0	1561.0	1559.0
1.1	1713.9	1713.9	1709.3
1.2	1866.8	1866.8	1859.6

Figure 3.12: The effect of different γ values on complexity of partitions of macc graph on a 16 processor crossbar

$O(K \log K + KN)$. In Algorithm-2, again the sorting takes $O(K \log K)$ time. For each of the K values of i, we update the values of *tavg* and *tvar* in constant time and evaluate the loop over N values of p. Within this inner loop, computation of LD(d, p) and F(D, p) can be done in constant time, but CCC(d, p) requires $O(K)$ time because it involves a summation over all vertices d_j. Since all the N iterations of the inner loop over p takes $O(KN)$ time, the time required to complete all the K iterations over i is $O(K^2N)$. This time clearly dominates the sorting time. Thus Algorithm-2 has a time complexity $O(K^2N)$. Algorithms 3 and 4 have the same complexity that may be evaluated as follows. Note that every time there is an unassigned vertex, all such vertices take part in the computation at that iteration. Hence, the loop over d in these algorithms is run K times in the first iteration, $(K-1)$ times in the second, etc. Thus this loop is evaluated $O(K^2)$ times. Within each of these loops, we do an inner loop over p, find value of $\psi(d)$ and evaluate the Merit function. Of these three actions, the Merit function evaluation requires only a constant time. Evaluation of $\psi(d)$ (and of $\psi'(d)$ in Algorithm-4) requires $O(N)$ time. Finally, the evaluation of the N cycles of the loop over p takes $O(KN)$ time because in each cycle, the LD and F functions evaluate in constant time and CCC requires $O(K)$ time. Clearly, of the three actions, this is the most time consuming one. Therefore the total time of Algorithm-3 and Algorithm-4 is $O(K^3N)$.

As evident from Figures 3.6-3.8 that Algorithm-4 outperforms the other three algorithms for a large range of λ. In particular, when $0.4 < \lambda < 1$, Algorithm-4 does definitely show promise in all architectures. One may note here that simulation task, in general, is communication intensive. Therefore one would expect λ values in the neighborhood of 0.5 where Algorithm-4 is better. In most optimization problems,

one would expect maximizing the rewards at each iteration to yield the best possible solution. In the case of the assignment problem, this corresponds to Algorithm-3. However, results indicate that Algorithm-4, which minimizes the wrong selection penalty at each iteration gives much better results. This is noteworthy in light of the fact that both these algorithms have identical time complexities. Simulation is a highly time consuming task. On the other hand, designing the partitions to be used in parallel simulation is a relatively light task that needs to be done only once statically before the actual simulation begins. Thus the time spent on obtaining a better partition is well spent. From this point of view, even though Algorithm-4 has a higher complexity compared to Algorithms 1 and 2, its use is still justified because of the better partitions it produces.

The experiments reported in the previous section were carried out for three architectures of same cardinality: a degree 4 hypercube, a 4×4 grid and a 16 processor crossbar machine. The performance of partitions obtained by Algorithm-4 on these architectures is shown in Tables 3.2-3.4. Three values of design parameter γ were used: 0.5, 0.8 and 10. This data indicates two important trends. Firstly, it may be noted that on each architecture, the results obtained for different γ values are nearly identical. This illustrates the fact that the final results obtained from Algorithm-4 are relatively insensitive to small variations in γ. In a discrete event simulation environment, the value of γ is domain specific and should be estimated to the best of ones ability. However our results indicate that Algorithm-4 can tolerate some errors in γ. Secondly, one can see that the results of partition complexity given in Tables 3.2-3.4 are architecture dependent. In order to understand the reasons for this, one needs to examine the communication properties of these three networks described below.

The N processor hypercube has degree $\log N$. Each processor in this hypercube has $\log N$ neighbors and the total number of bidirectional links is equal to $(N/2)\log N$. Having a small number of neighbors and a small number of links is necessary for realizability. The diameter (worst distance between any pair of processors) of this network is also only $\log N$. Thus from communication viewpoint, hypercube is a highly desirable network. The N processors of a grid are arranged as an $\sqrt{N} \times \sqrt{N}$ array. There are $(\sqrt{N}-1)^2$ links in a grid and the degree of the nodes ranges between 2 and 4. The diameter of the grid is $2(\sqrt{N}-1)$. The N processors in a crossbar switch are connected as a complete graph. Thus the degree of each node in a crossbar switch is $N-1$ and there are $N(N-1)/2$ bidirectional links. The diameter of the network is 1. The $O(N^2)$ links makes the crossbar a highly undesirable network from implementation standpoint unless N is very small. Since the crossbar has the best possible communication performance (distance between any pair of nodes = 1!), the performance obtained for the crossbar is the best and constitutes the lower bound. The hypercube interconnect scheme is not as good as a crossbar, but still is richer than that of a grid and therefore its performance is better than that of the grid. However if the parallel processor is to be implemented as a VLSI circuit as in a hardware accelerator, the layout complexity is an important criteria. The grid has the least and the crossbar, the most layout overhead.

3.6 Summary

This chapter is concluded by noting that in simulation assignment applications, one deals with rather irregular task graphs. The assignment algorithms are hueristic and their performance is measured by the computational and communication load

on each resultant partition, rather than throughput. It is shown that the proposed algorithm (Algorithm-4) based on minimizing the penalties is highly superior to other algorithms which maximize the rewards. This is in spite of the fact that this new algorithm does not have a higher computational complexity than some others. This algorithm is also rather insensitive to the only design parameter γ that the user is supposed to estimate. Finally, it should be mentioned that this superior performance of the algorithm holds good on all popular architectures.

Chapter 4

Exploiting Functional Parallelism in Simulation

4.1 Introduction

Parallel processing has been successfully applied to improve system performance in a variety of applications. In this work, we develop a parallel discrete-event simulation algorithm with multiple instruction multiple data (MIMD) paradigm in mind. The proposed algorithm is formally verified using petri net [30–32] models. However, the main emphasis of this chapter is in investigating low level functional parallelism inherent in the algorithm. Typically, $algorithm-to-architecture$ development is an iterative process that may go through several stages of refinement. To come up with the best solution, one has to use statistical performance modeling techniques to fully analyze system performance. Stochastic petri net theory [33] is used to capture the expected performance of the proposed architecture. In Section 4.2, petri-net theory is presented. It contains the mathematical preliminaries required in the rest of the chapter. The proposed cone-based parallel simulation algorithm and its

formal verification is explained in Section 4.3. Finally, Performance of the proposed architecture is predicted in the last two sections.

4.2 Overview of Petri-Net Theory

Petri-Nets (PN) are a graphical and mathematical tool applicable to many systems. They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, PN can be used as visual communication aid similar to flowcharts, block diagrams and networks. In addition, tokens are used in these nets to represent the dynamic and concurrent activities of the systems. Historically speaking, the concept of PN was first introduced by Carl Adam Petri as a part of his dissertation to the faculty of Mathematics and Physics at the Technical University of Darmstadt, Germany [30]. Since then, many researchers have introduced significant additions to this theory [33,34] that have been applied to various aspects of computer systems or algorithm modeling very successfully [35–37].

A PN consists of

- a set of places P (drawn as circles),
- a set of transitions T (drawn as bars), and
- a set of directed arcs A.

Arcs connect transitions to places and places to transitions. Places may contain tokens. A number inside a place represents the number of tokens in that place. The marking, or state of a PN, is defined by the number of tokens contained in each place and is denoted by a vector M, $M = \{m_1, m_2, m_3, ..., m_{|P|}\}$, whose i^{th} component represents the number of tokens m_i in the i^{th} place. The construction of a PN model

requires the specification of the initial marking M_0.

A place p_i is an input to a transition t_j if atleast one arc exists from the place to the transition and is denoted by $I(p_i, t_j) > 0$. A place p_k is an output from a transition t_j if atleast one arc exists from the transition to the place and is denoted by $O(t_j, p_k) > 0$. An inhibitor-arc, $H(p_i, t_j) > 0$, is represented by a small circle in one end. It is particularly useful in representing negated logic that participates in enabling a transition. The multiplicity of arcs are denoted by a slash accompanied by a number in the arcs. This PN extension was introduced to compactly represent parallel arcs connecting the same *place−transition* pair. A transition is enabled when each of its input places contains at least as many tokens as the multiplicity of arcs and at most one less token than the multiplicity of inhibitor arcs, i.e. a transition t_j is enabled, iff $m_i \geq I(p_i, t_j)$, $\forall i$ *s.t.* $I(p_i, t_j) \geq 0$ and $m_k < I(p_k, t_j)$, $\forall k$ *s.t.* $H(p_k, t_j) \geq 0$. An enabled transition can fire. The firing of a transition removes all enabling tokens from its input places and deposits tokens to its output places. The number of tokens deposited are equal to the multiplicity of arcs connecting these places. An inhibitor arc neither removes nor deposits any token. The firing of a transition modifies the distribution of tokens in places and thus produces a new marking of the PN.

For an initial marking M_0, the reachability set $R(M_0)$ is defined as a set of all markings that can be reached from M_0 by a sequence of transition firings. If the PN is such that, for any possible marking, the number of tokens in any place is less than or equal to k, for some integer k, then the PN is said to be $k-$*bounded* (conservative if exactly equal to k). Also, if for any marking $M_j : M_j \in R(M_0)$, and for any transition $t_i \in T$, there exists a transition firing sequence starting from M_j and ending on a marking $M_l : M_l \in R(M_0)$ at which t_i is enabled, the PN is said to be *live*.

A stochastic Petri net (SPN) is a PN where each transition is associated with an exponentially distributed random variable that expresses the firing rate of the transition. Due to the memoryless property of the exponential distribution of firing rates, it has been shown [33] that the reachability graph of a bounded SPN is isomorphic to a finite Markov chain. While Markovian theory is a powerful tool for predicting system performance, often times it is not easy to come up with a Markov model that accurately represents complex system behavior. It is in this situation, practitioners and theoreticians alike would find SPN to be a very useful way of synthesizing Markov models. A formal definition of $SPN = (P, T, I, O, H, L)$ with an initial marking M_0 is thus the following:

$P = \{p_1, p_2, p_3, ..., p_n\}$ is a set of places,

$T = \{t_1, t_2, t_3, ..., t_m\}$ is a set of transitions,

$I : \{P \text{ X } T\} \rightarrow N$,

$O : \{T \text{ X } P\} \rightarrow N$ and

$L = \{l_1, l_2, l_3, ..., l_m\}$

is a set of firing rates such that l_j is the rate of exponential distribution associated with the firing time of t_j. l_j is said to be marking dependent when it is a function of current marking M_i. The transition rate from state i (corresponding to marking M_i) to state j (M_j) is $q_{ij} = \sum_{k \in U_{ij}} l_k$, where U_{ij} is the set of transitions enabled by marking M_i, whose firing generates M_j.

SPNs have been extended to a class of generalized stochastic Petri nets (GSPN) [38,39] in order to cope with the state space explosion problem. PN transitions that have firing rates, several order of magnitude higher than the rest, are considered as having infinite firing rates. This approximation has negligible impact on the perfor-

mance measurement accuracy but dramatically reduces the state-space size [38]. In a GSPN, the set of transitions T comprises two different classes of transitions. These are immediate and timed transitions. Immediate transitions fire in zero time once they are enabled. Timed transitions fire after a random, exponentially distributed enabling time. GSPNs are logical extensions of SPN, where immediate transitions represent transitions with infinite firing rate. Exactly as with SPN, several transitions may be simultaneously enabled by a marking. The enabled transition fires with a probability $\frac{l_i}{\sum_{k \in U} l_k}$, where the set of enabled transitions comprises U and l_i is the firing rate of the transition which fires. Clearly, if U comprises one immediate transition and several timed transitions, then the immediate transition will fire with probability 1. However, for any marking at which several immediate transitions are enabled, they fire with equal probability unless a probability distribution over the enabled transition selection is specified. The probability distribution is called random switching distribution and the set of immediate transitions is said to form a random switch.

The analysis of GSPN is based on characterizing the underlying process as a stochastically discontinuous Markov process. The process transits from one tangible state to another with intermediate visits to vanishing states. The analysis of this process is made by recognizing an Embedded Markov Chain (EMC) within the process. In an EMC, a discrete time Markov chain (DTMC) is embedded within a continuous time Markov chain (CTMC). The DTMC is obtained by considering both immediate and timed transitions in the reachability graph of the GSPN. The DTMC determines a partition of the state space in terms of tangible-states and vanishing-states. Each tangible-state consists of one marking or several vanishing markings. From the anal-

ysis of this Markov chain we can obtain the transitional probabilities from vanishing state to tangible-states as well as visit-ratios of each tangible-state. By aggregating the vanishing-states into corresponding tangible-states, a CTMC is obtained. The state-space of this CTMC is defined on the set of tangible-states. Steady state probabilities of the states can then be evaluated by analyzing the CTMC. It provides a measure of the fraction of the cycle-time spent at a particular steady-state. Linear (weighted) functions of the CTMC steady-state probabilities provide interesting performance indices.

4.3 Parallel Simulation Algorithm

Simulators can be classified according to the type of internal model they process. In *event driven* simulation, only active portions of the circuit are evaluated every cycle. A device in a circuit is active if it has atleast one input event. In order to propagate events, such simulators need to maintain dynamic event-lists that are expensive to manage and require development of complex customized data structures covering all event infectable devices. A *compiled* simulator on the other hand executes pre-compiled code created by mapping the structural definition directly into machine code. Evaluation is fast but not optimized. In each cycle, the entire circuit is evaluated and so one does not have to worry about managing events. For plain levelized logic, this approach definitely works much faster than its event driven counterpart, but tracking synchronous loops or memory is expensive. Such loops are temporally expanded into iterative logic arrays to force levelization.

The proposed parallel simulation approach is a hybrid of event-driven and compiled simulation technique. Rankable (feedback-free) portions of the design are ab-

stracted as cone and are evaluated at every simulation tick. The only event-flow recorded is *between* blocking-devices (sequential elements in a design). Such recording helps ascertain blocking-devices whose output is expected to change. Our statistical analysis of several large circuits show that in a typical synchronous design, number of blocking devices is only 10-15% of the total device count. Thus keeping track of changes at the output of blocking devices and communicating these changes to the subsequent cones of combinational logic is a relatively simple task. Further, the approach fully extracts the parallel components within a data-set which may be evaluated concurrently (without any mutual interference) in any simulation tick. Finally, it also allows for a coarser granularity which is better suited to modern parallel machines which have extremely fast computational capabilities but relatively slow and with constant overhead interprocessor communication facilities.

Synchronous digital networks can be simulated by keeping track of two lists. The first is an *active signals* list (L1), that is processed by updating signal (pending) changes and scheduling active blocking-devices. The second is an evaluation list (L2) that contains all blocking-devices whose output must be recomputed because one or more of its inputs has changed. This list is processed by updating scheduled input changes, evaluating and subsequently scheduling new signal changes. The processing of L1 populates L2 and conversely, processing of L2 generates fresh L1. Besides separating the two conceptually different tasks of simulation – evaluation and scheduling of lists L1 and L2 also bring out the inherent concurrency in simulation. $(k+1)_{th}$ simulation time-frame L1-computation can be performed in parallel with k_{th} simulation time-frame L2-computation. This is only possible because L1-computation is several orders of magnitude lighter than the L2-computation. If incrementally generated sig-

nal schedules are passed directly (through dedicated buses) to the processors devoted to subsequent cones, L1-computation can get started. Thus, $(k+1)_{th}$ simulation time-frame L1-computation will be completed soon after k_{th} simulation time-frame L2-computation has completed.

The proposed simulation algorithm proceeds as follows:

step-1: Global time manager sends out primary input events (if any) and a signal to all processors to begin a new simulation clock tick.

step-2: After receiving signal from the time manager, each of the L1 and L2 processors start working on their scheduled list of events.

step-3: After L2 processing is completed, its adjacent L1-processor communicates *new* signal event status to *all* other L1 processors in the network.

step-4: Each L1-processor responds to messages from its peers, iff the signal events happen to fanout to any of the blocking devices assigned to the L1-L2 processor group.

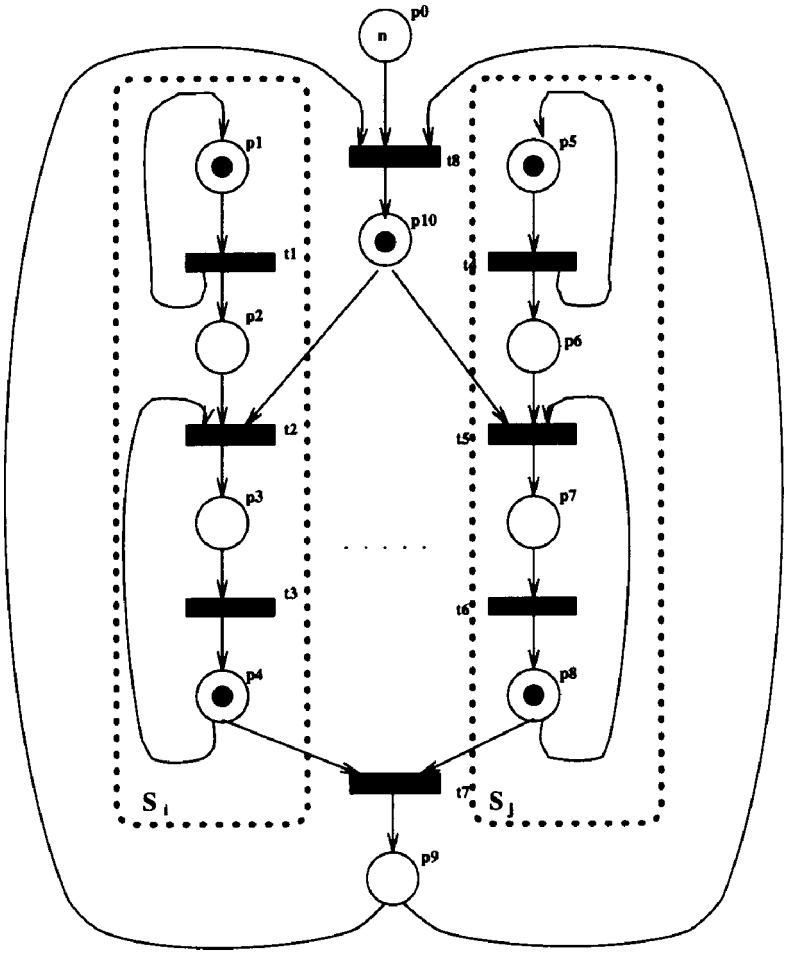
step-5: After receiving messages from all its peers and processing any incremental entries, each L1 processor sends out a *done* message to the global time manager.

step-6: After receiving *done* messages from all L1-processors, the time manager advances the simulation tick and returns to step-1.

We now derive the PN model for the proposed algorithm. Each subnet s_i of the PN shown in Figure 4.1 illustrates L1-L2 processing cluster. Transition t_1 in subnet s_i represents start of L1 processing and t_2 represents *wait* for L1 processing to complete. Places around the transitions serve as pre-condition and post-condition. For example, place p_1 represents unprocessed L1 events. Place p_2 represents the completion of that task. Transition t_3 represents the processing of blocking devices and place p_3

unprocessed L2 events. t_3 can be further expanded into another subnet of finer time-scale resolution as shown in the next-section. Place p_4 marks the end of L2-processing for the current simulation cycle. This also enables subnet s_i to communicate its new L1 status to the rest of the synchronization network. Transition t_7 represents signal synchronization and p_9 marks its completion. Similarly, t_8 represents global time management and p_{10} marks the beginning of a new simulation cycle. Note that transition t_8 can fire iff there is atleast one token in place p_0, meaning that the simulation stops when the time limit of simulation is reached. For each simulation cycle, place p_0 looses a token and is therefore one step closer to completion.

As indicated in Chapter 3, one can take advantage of data-parallelism by having subnets similar to s_i work in parallel. The number of such parallel subnets is bounded by the cost of synchronization. Communication slowdown due to multiple subnets is discussed in the next Chapter. A simulation cycle is marked *completed* after each subnet has received messages from all its peers *and* sent a *done* message to the global time-manager. A new simulation cycle then begins with initialization which requires distributing primary-input changes to individual subnets and advancing the simulation clock. The simulation loop will continue *until* there is atleast one token in place p_0 (**completeness** property). Initial-marking for the system is defined by tokens in places defining *unprocessed L1 entry*, *end of L2 processing* and place p_{10}. Within each subnet s_i, every state of the reachability-set has a path back to the initial marking. Also, a faster subnet may temporarily wait for a slower subnet to finish its computation, but the synchronization cannot get suspended forever. Therefore, one can say that the PN is free from **deadlocks**.



t1,t4 -- start L1-processing

t2,t5 -- end L1-processing

t3,t6 -- L2-processing

t7 -- Synchronize signal changes

t8 -- time synchronization

p0 -- simulation time-frames

p1,p5 -- L1-entry

p2,p6 -- finished processing L1-entry

p3,p7 -- L2-entry

p4,p8 -- finished processing L2-entry

p10 -- cycle initialization

p9 -- signal synchronization completed

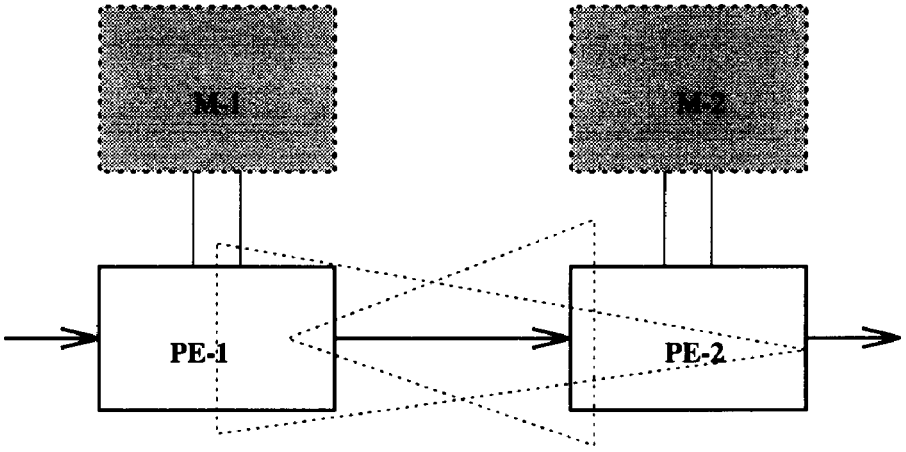
Figure 4.1: PN model for parallel discrete event simulation

4.4 2-Dimensional Funneled Pipeline Algorithm

The computational load of the algorithm in Figure 4.1 is concentrated in the step to recompute outputs of blocking devices, whose at least one cone input has changed. The computation of each blocking-device proceeds as follows. First, control-cones of the device are evaluated. If atleast one control cone has an event at the output, then the corresponding data-cone guards (horn-clause) are evaluated. The assertion of these guards trigger the evaluation of individual data-cones. For example, a simple blocking device such as J-K flip-flop may have control cones evaluating its clock, set and reset inputs and only the clause $(clock \wedge (\neg set) \wedge (\neg reset))$ being true, does one need to evaluate cones at J and K inputs. Our statistical analysis shows that in most realistic VLSI circuits, the data cones have a much longer depth and consequently computational complexity compared to control cones. The procedure outlined here allows one to avoid computation of data-cone if the assertion test fails. Note that while simulating a large circuit, multiple blocking devices are mapped onto the same processing node by algorithms developed in Chapter 3. The sequential steps associated with each blocking device (computation of control-cone with the horn-clause and the computation of data cones) may therefore be pipelined. Thus the computational part of the conceptual processing node may in fact be a two stage pipeline made up of $PE1$ and $PE2$ as in Figure 4.2. Thus the prioritized evaluation algorithm can be realized as an *asynchronous* funneled pipeline algorithm. We call it a funneled pipeline algorithm because at each stage, guard for the succeeding stage is evaluated and the computation-load is forwarded further only if the assertion is true. The shrinkage of computation data space varies from one stage to another, but can be considered to be constant over simulation time-frames. The pipe has two interesting

characteristics. The shrinkage of computation data-space forms a forward funnel and also the complexity of each task grows with the depth of the pipe (reverse funneling). This concept of funneling pipeline is a very general one and may be applied to many other situations involving asynchronous pipelines with possible deaths along the pipe. In our situation, the death-rate at the control-cone evaluation stage is minimal. So, for performance evaluation, it has been merged with horn-clause evaluation. Similarly, the data-cone evaluation and signal scheduling step can be merged into one. As a result, we have a two-stage 2-dimensional funneled pipeline algorithm. Memory is partitioned so that each stage can independently work on its own memory segment. In stage-1 memory, compiled code for control-cones are stored along with *current* fan-in values and horn-clause definitions. In the second stage memory, data-cone evaluations are stored along with the *current* fan-in values and fan-out signal table.

To predict the performance of the pipeline, we create a GSPN model of the system, as shown in Figure 4.3. Each stage of the pipe has an input buffer and an output buffer. Input buffer of stage-1 is always assumed to be full. Similarly, output buffer of stage-2 is always assumed to be empty. This merely implies that there is always new data available for processing and the results are instantly flushed from the pipeline. Transition t_4 represents the flushing of the pipe due to death in stage-1. Such flushing takes place when all the horn-clauses (for a blocking-device computation) evaluate to false. Let α represent the survival probability of blocking-device computations from stage-1 to stage-2 and μ_1 and μ_2, the exponentially distributed firing rates of transitions t_1 and t_2 respectively. Copying (transition t_3) the contents of output buffer of stage-1 to input buffer of stage-2 is a relatively faster operation and is therefore represented by an immediate transition with infinite firing rate. Tokens in places



PE-1: Control-cone, Horn-clause Evaluation
PE-2: Data-cone Evaluation
M-1: Stage-1 Memory
M-2: Stage-2 Memory


Complexity Funneling


Computation Data Space Funneling

Figure 4.2: 2-dimensional funneled pipeline data-path

p_1, p_2, p_5 *and* p_6 represent the initial marking of the PN. Note that places p_1 and p_6 always hold tokens because of our assumptions. Reachability graph ($R(M_0)$) of the PN can be computed as follows. The firing sequences, $M_0\ t_1\ M_1\ t_3^{\alpha}\ M_2\ t_2^{\beta}\ M_0$ and $M_2\ t_1^{\theta}\ M_3\ t_2\ M_2\ t_2^{\beta}\ M_0$ indicate that all the markings M_1, M_2 *and* M_3 are reachable from M_0. This reachability graph is shown in Figure 4.4. We now prove the following theorems to demonstrate that the PN has the necessary properties.

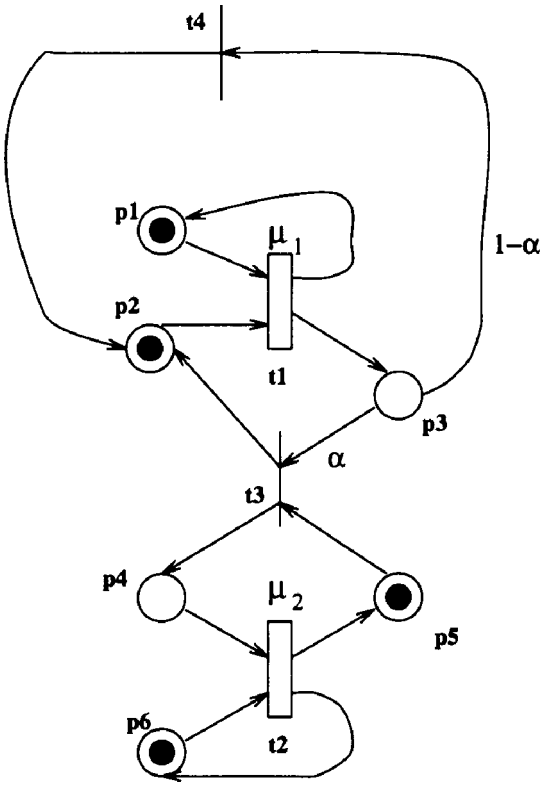
Theorem-1: The PN is safe for M_0.

Proof: In Figure 4.4, $m_{|p|} \leq 1$ for each p in P and for any marking M reachable from M_0. Hence the PN is safe for M_0. ∎

Theorem-2: The PN is conservative.

Proof: In Figure 4.4, p_1 and p_6 always have a token. Also, $p_2 + p_3 + p_4 + p_5$ is always 2. That means the token count is constant (4) for all markings. Hence the PN is conservative. ∎

Discrete time Markov chain (DTMC) of the reachability graph of Figure 4.4 is shown in Figure 4.5. It may be explained as follows. Transition from M_1 to M_2 takes place only if the computation-load survives at stage-1 and therefore has a probability α. With the rest of the probability $1 - \alpha$ one goes from M_1 to M_0. M_1 is a vanishing state because only immediate transitions are active in this state. In M_2, both transition t_1 and t_2 are active, where probability of t_2 firing is $P\{t_2\} = \frac{\mu_2}{\mu_1+\mu_2} = \beta$ and $P\{t_1\} = \frac{\mu_1}{\mu_1+\mu_2} = \theta$. If t_2 fires, the process transits from M_2 back to M_0. But, if t_1 fires, next-state is M_3 if the computation-load survives stage-1. Otherwise, the process remains in state M_2. The transition rate diagram of the continuous time Markov chain (CTMC) is formed by aggregating the vanishing state M_1 with the



p1: Stage-1 input buffer full
p2: Stage-1 output buffer empty
p3: Stage-1 output buffer full
p4: Stage-2 input buffer full
p5: Stage-2 input buffer empty
p6: Stage-2 output buffer empty
t1: Stage-1 computation
t2: Stage-2 computation
t3: copy out-buffer (stage-1) to in-buffer (stage-2)
t4: flush pipe

Figure 4.3: GSPN model of the pipelined data-path

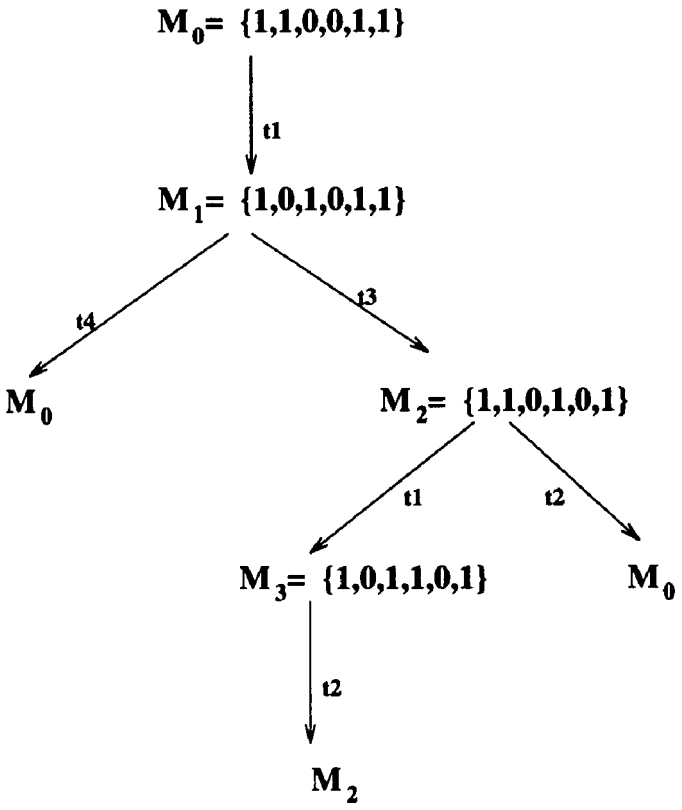


Figure 4.4: reachability graph of the GSPN model

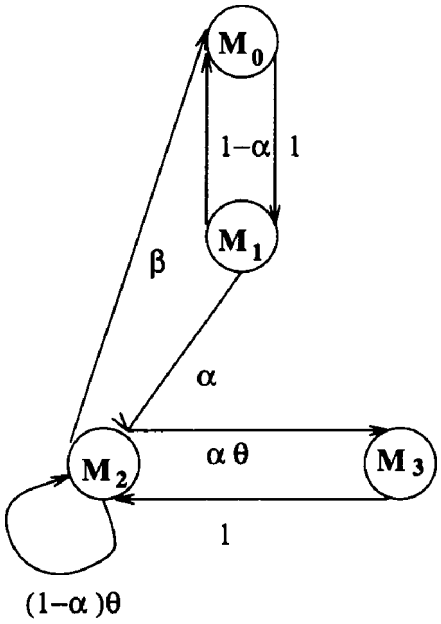


Figure 4.5: DTMC of the EMC for the GSPN model

tangible state M_0. This is shown in Figure 4.6. Note that vanishing states do not exist in a continuous time Markov chain. Let $\pi_0^{(c)}$, $\pi_2^{(c)}$ and $\pi_3^{(c)}$ in the CTMC denote the steady state probability of being in the states 0, 1, 2 *and* 3 respectively. Similarly in the DTMC, $\pi_0^{(d)}$, $\pi_1^{(d)}$,$\pi_2^{(d)}$ and $\pi_3^{(d)}$ denote the steady state probability distribution of the respective states. It can be shown that the MC is irreducible, aperiodic, time-homogeneous and positive-recurrent [40]. Therefore it is ergodic and the steady-state probability distribution is identical to stationary probability distribution.

For a DTMC with one step transitional probability matrix P_M, one can get the steady-state probability distribution by solving the equations

$$\sum_i \pi_i^{(d)} = 1 \; and$$

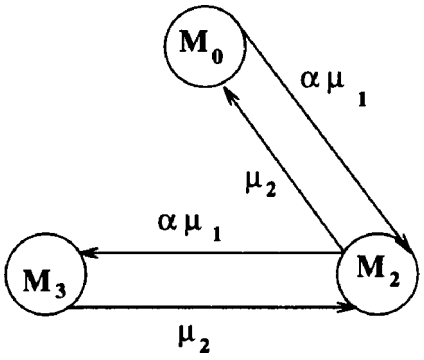


Figure 4.6: CTMC of the EMC for the GSPN model

$$\begin{pmatrix} \pi_0^{(d)} \\ \pi_1^{(d)} \\ \pi_2^{(d)} \\ \pi_3^{(d)} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1-\alpha & 0 & \alpha & 0 \\ \beta & 0 & (1-\alpha)\theta & \alpha\theta \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \pi_0^{(d)} \\ \pi_1^{(d)} \\ \pi_2^{(d)} \\ \pi_3^{(d)} \end{pmatrix}$$

$\pi_0^{(d)} = \frac{\beta/\alpha}{1+\beta/\alpha+\alpha\theta}$,

$\pi_1^{(d)} = 0$, since M_1 is a vanishing state,

$\pi_2^{(d)} = \frac{1}{1+\beta/\alpha+\alpha\theta}$,

$\pi_3^{(d)} = \frac{\alpha\theta}{1+\beta/\alpha+\alpha\theta}$.

It can be shown that for a CTMC,

$$\frac{d\pi_i^{(c)}}{dt}(t) = \sum_{j \in S} q_{ji}\pi_j(t),$$

where q_{ji} is the transition rate from state j to state i [41]. Since, steady state probability distribution exists for the CTMC (ergodic), we have

$$\frac{d}{dt}\pi_i^{(c)}(t) = 0.$$

Thus one gets,

$$\pi_2^{(c)}\mu_2 = \pi_0^{(c)}\alpha\mu_1,$$

$$\pi_2^{(c)}\alpha\mu_1 = \pi_3^{(c)}\mu_2.$$

Also the sum of these steady state probabilities is 1. Solving these relations, one can get the following values of the continuous time probabilities.

$\pi_0^{(c)} = \frac{1}{1+\rho+\rho^2}$,
$\pi_2^{(c)} = \frac{\rho}{1+\rho+\rho^2}$ *and*
$\pi_3^{(c)} = \frac{\rho^2}{1+\rho+\rho^2}$,
where $\rho = \alpha\frac{\mu_1}{\mu_2}$

One important parameter that provides meaningful information about the efficiency of the asynchronous algorithm implementation is the *processor utilization*. **Processor Utilization** (PU) can be defined as,

$$PU = \sum_i p_i\pi_i^{(c)},$$

where p_i represents the number of processors active in state i. Clearly, the ideal value of PU is the total number of processors in the architecture. However, since all processors are not used in all the states, it is generally less than that. In our case, states M_0 and M_3 are *inefficient* because only one processor out of 2 is in use in these states (in M_0, first processor and M_3, the second). On the other hand, both processors are exploited during M_2. By weighing the processor utilization in each state, one gets in our case,

$$\begin{aligned} PU &= \pi_0^{(c)} + 2\pi_2^{(c)} + \pi_3^{(c)} \\ &= 1 + \frac{\rho}{1+\rho+\rho^2} \end{aligned}$$

This processor utilization equation allows one to employ design tradeoffs. For a given death-rate α, depending upon the relative values of the mean firing rates μ_1 and mu_2 of the two tasks of evaluating the control-cone plus horn-clause and computing data-cone, the PU may change drastically. Thus if $\mu_1 \ll \mu_2$ ($\rho \simeq 0$) or $\mu_1 \gg \mu_2$ ($\rho \simeq \infty$), PU tends to be 1. But μ_1 and μ_2 can be altered either by using more powerful processors or by using multiple processors in place of a single conceptual pipeline processor. Thus their ratio ($\frac{\mu_1}{\mu_2}$) can be varied to suit the survival rate α. By mathematical manipulation one can show that our PU expression achieves a maximum of 1.33 when $\rho = 1$. Note, $\rho = 1$ implies that in order to extract maximum performance, the relative ratio of complexities, $\frac{\mu_1}{\mu_2}$, must exactly match $\frac{1}{\alpha}$.

To get an idea of the survival-rate parameter α, simulation on real circuits were performed using a software simulator. Transitions on cone inputs and output were monitored. Our experiments indicate that on an average, for a blocking-device (BD) computation with data and control cone input events, only 40-60% of the time horn-clause for the BD is asserted. This implies that typically the α value ranges from 0.4 through 0.6. A graph of PU versus α is shown in Figure 4.7. For curves corresponding to relative processing speeds of 5 and 8, the processor-utilization degrades rapidly in the effective α band mentioned above. However, the performance curve corresponding to relative processing speed of 2 has the desirable property of sustained peak performance in this band. This analysis is interesting because it imposes a constraint on the hardware design of the processors. The restriction again is more on the relative ratio of speeds rather than absolute processing speeds.

A simulation program was written in $C++$ to mimic the operation of the pipeline strategy defined above. Each job entering the pipeline is assigned complexities based

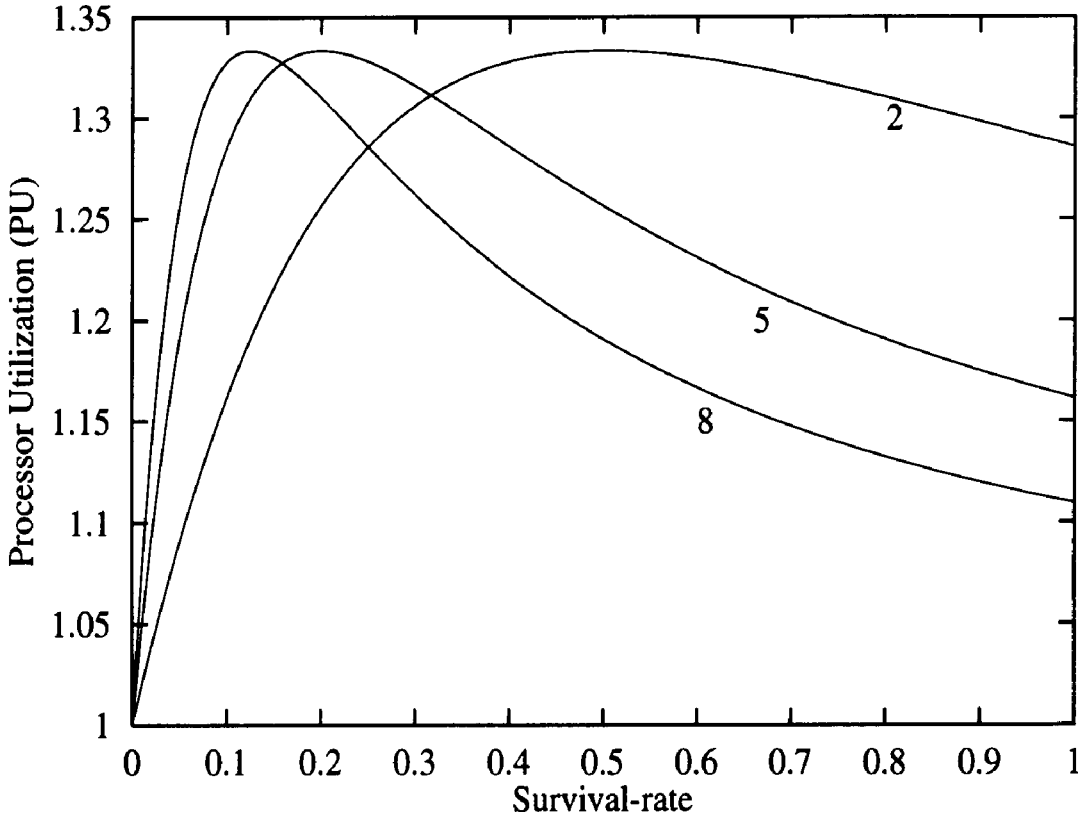


Figure 4.7: Processor utilization versus survival-rate

on an exponential distribution. For a given probability of survival (from stage-1 to stage-2 of the pipeline) $\alpha = 0.5$ and different mean speed ratios, $\frac{\mu_1}{\mu_2}$, the maximum processor utilization converged to 1.29. That means, our analytical results have an error of less than +4%. Clearly this validates our analytical results.

For a CTMC, waiting time in a tangible-state is exponentially distributed. The probability density function of the waiting-time distribution can be written as [42],

$$f = -q_{ii}e^{q_{ii}w},$$

where $-q_{ii}$ is the transition rate of moving out of state i. Therefore, the average waiting time $E[W_i]$ in state i is given as

$$\begin{aligned} E[W_i] &= \int_0^\infty -wq_{ii}e^{q_{ii}w}dw \\ &= -\frac{1}{q_{ii}}\int_0^\infty ue^{-u}du, \ where\ u = -q_{ii}w \\ &= -\frac{1}{q_{ii}}\Gamma(2) \\ &= -\frac{1}{q_{ii}} \end{aligned}$$

It can be shown that q_{ii} (rate at which process transits out of state i) is equal to the sum of the firing rates of transitions active in state i [33]. Average waiting times in states M_0, M_2 *and* M_3 are therfore given as $\frac{1}{\alpha\mu_1}$, $\frac{1}{\alpha\mu_1+\mu_2}$, *and* $\frac{1}{\mu_2}$ respectively.

The average speed of a processor system can be measured by its throughput. **Throughput** in this context is defined as the rate at which jobs are processed by the two stage pipelined processor system. Some of the jobs in our funneled pipeline algorithm may not survive to proceed to the second stage while others get processed in both the stages. Since in either case, all jobs are processed in the first stage, the

throughput of the system is given by the average rate of completion in this stage. As illustrated in the Petri net diagram and reachability graph, stage-1 is active in states M_0 *and* M_2. Therefore, the throughput of the two stage pipeline is given as,

$$\begin{aligned} T_{2D_{funnel}} &= (\pi_0^{(c)} + \pi_2^{(c)})\mu_1 \\ &= \frac{1+\rho}{1+\rho+\rho^2}\mu_1 \\ &= \frac{(\mu_2/\alpha)^2 + \mu_1^2}{\mu_1\mu_2/\alpha + (\mu_2/\alpha)^2 + \mu_1^2}\mu_1 \\ &= \frac{xy^2 + x^2y}{x^2 + y^2 + xy}, \end{aligned}$$

where $x = \mu_1$ and $y = \frac{\mu_2}{\alpha}$. Note that the function is symmetric with respect to x and y. Also, increasing x unboundedly makes the function dependent on y and vice versa. Mathematically, it can be shown that the system throughput increases unboundedly only when $x = y$. In all other cases, the value of the function is determined by the minimum of x and y. The design implication of this analysis is that for a given survival rate α, the average relative speeds of the two stages is fixed and is given as,

$$\mu_1 = \frac{\mu_2}{\alpha}.$$

4.5 1-Dimensional Funneled Pipeline Algorithm

As illustrated by subnet s_i of Figure 4.1, computation in each simulation time-frame can be partitioned into two segments – L1-processing and L2-processing. Algorithm level parallelism can be realized by allowing both the steps to progress simultaneously. L2-processing is generally more expensive than L1-processing. To extract maximum throughput, the processing speeds should be made comparable. To do this, L2-processing load can be distributed among a set of homogeneous PEs. Assuming that

L2-processing complexity is roughly m times that of L1-processing, and that there is a queuing facility of limited size available at L1-processor, one would need several ($m > 1$) processors for L2 processing. The parallel simulation algorithm can therefore be configured as a two-stage, 1-dimensional funneled pipeline algorithm.

An animated view illustrating the computational dynamics of the L1-L2 processor-cluster is shown in Figure 4.8. In our discussion, we assume the value of m to be 2. In frame-1, the L1-processor incrementally adds any new blocking-device (BD) schedule resulting from primary input events for that simulation cycle. Note that primary input events (if any) are distributed by the global time-manager along with the *start* message for the simulation cycle. Blocking-device computation workload is then distributed to individual L2 processors through dedicated buses. In frame 2, both the L2 processors simultaneously work on their scheduled BDs. New signal event generated as a result of change on the blocking device output is dispatched back to the L1 processor before the L2 processor can start processing the next BD in its list. A filled circle with an arrow pointing back to L1 processor indicates that a signal event is scheduled as a result of blocking-device computation. Hollow circle indicates, no signal event resulted from that computation. The blocking device computation either expired at stage-1 of the pipeline or failed to produce an event after stage-2 computation. In frame 3, each L2 processor works on a blocking-device but now on top of that, the L1 processor processes the new scheduled signal event. This strategy allows all the processors to work in parallel. The processing speeds of L1 processor and L2 processors need to be matched in such a way that there is no workload overflow in the L1 processor's native buffer. Frame 4 illustrates an interesting scenario – neither of the L2 processors schedule new signal event. That

means the L1 processor has to idle out till new workload is shipped to it. Behavior of the system in frame 5 is identical to frame 3. In frame 6, both the L2 processors have completed processing their workload and the simulation cycle ends soon after the L1 processor finishes processing its remaining workload. Frame 1 activity overhead can be estimated deterministically and would constitute less than 5% of the total computation time per cycle. This is because, sending packets along dedicated buses and processing relatively sparse primary input events are fairly fast operations. The remaining 95% of the computation time can only be estimated using probabilistic analysis.

We now show how the petri-net (PN) model of this system can be used to predict the performance of concurrent L1 and L2 processing. In Figure 4.9, transition t_2 (representing L1 processing) has an exponentially distributed firing rate η. Places p_2 and p_3 correspond to conditions, L1 processor active and L1 processor waiting for new work respectively. Transition t_1 (representing L2 processing) has an equivalent firing rate λ_{eq}. Let m L2 processors each with an exponentially distributed firing rate of λ contribute to the resultant firing rate of λ_{eq} $(= m\lambda)$. Intuitively, λ_{eq} represents the rate at which workload piles up at the L1 processor's native buffer. The size of this buffer is finite. This means if a signal schedule arrives when the buffer space is full, L1 processor simply ignores the message and returns a negative acknowledgement to the sender. A positive acknowledgement means that the message was accepted. Place p_0 represents the full buffers and place p_1 represents the task in a buffer waiting for L1 processor to get free. Transition t_3 represents the start of L1 processing and is only enabled when L1 processor is available for computation and at least one task is waiting in its buffer. In this PN we assume the size of the buffer at L1-processor to

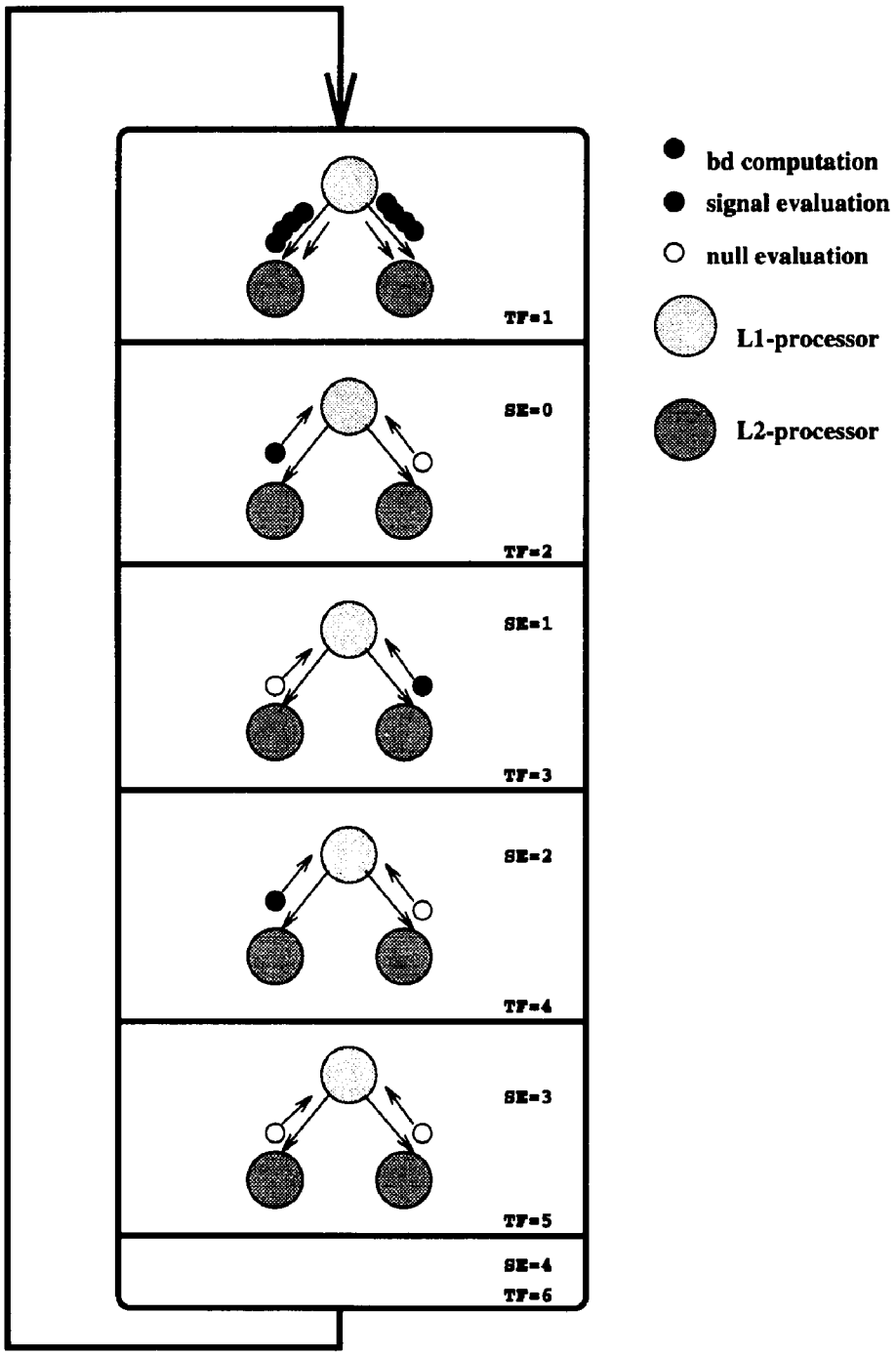


Figure 4.8: visual representation of architecture dynamics

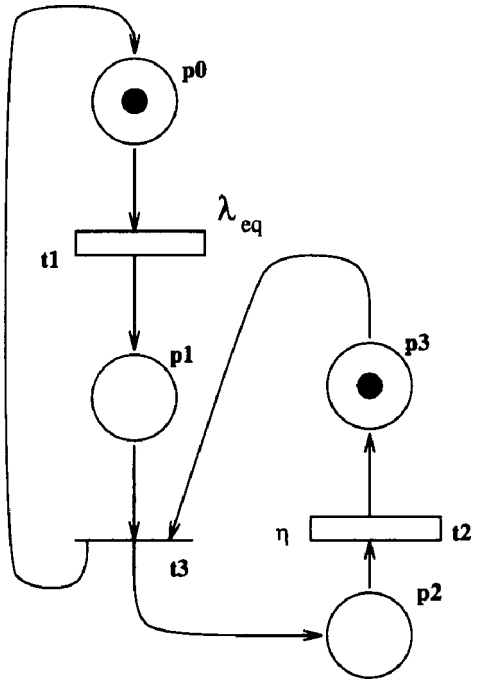


Figure 4.9: GSPN model for the 1-d funneled pipeline algorithm

be 1. The initial marking for the PN is defined by a token in place p_0 and another one in place p_2. A token in place p_0 implies that the buffer can hold atmost one request and one token in place p_2 represents a single L1-processor. Increasing either of the resources would mean increasing the initial number of tokens in the corresponding places by that much. However, the state space of the resulting reachability graph becomes very large and often times difficult to analyze.

The reachability-graph and CTMC (transition rate diagram) of the GSPN in Figure 4.9 are shown in Figure 4.10 and Figure 4.11 respectively. At steady-state, using the local balance properties and the same techniques as in the previous Section, we obtain the following solution,

$\pi_0^{(c)} = \frac{1}{1+m\sigma+m^2\sigma^2}$,

$\pi_2^{(c)} = \frac{m\sigma}{1+m\sigma+m^2\sigma^2}$ and

$\pi_3^{(c)} = \frac{m^2\sigma^2}{1+m\sigma+m^2\sigma^2}$

where $\sigma = \frac{\lambda}{\eta}$ and $\lambda_{eq} = m\lambda$.

The effective processing power of the L1-L2 processor-cluster is measured by the parameter, Processor Utilization (PU), defined in the previous Section. It is a sum of products, where each product term is the number of active processors in a state times the probability of being in that state and is given as follows.

$$\begin{aligned} PU &= m\pi_0^{(c)} + (m+1)\pi_2^{(c)} + \pi_3^{(c)} \\ &= m + \pi_2^{(c)} + (1-m)\pi_3^{(c)} \\ &= m + m\sigma\frac{1+m\sigma-m^2\sigma}{1+m\sigma+m^2\sigma^2} \end{aligned}$$

The graph in Figure 4.12 clearly shows that just increasing the number of L2 processors will not necessarily result in increased processing power. Each PU curve has a linear and a saturation region. Adding L2 processors increases the overall processing power of the system only upto the saturation point (crossover from linear to saturation region). Beyond this threshold point processor utilization is poor. When the ratio of mean processing speeds ($\frac{\lambda}{\eta}$) is either 0.5 or 0.8, saturation is reached very early. However, when the mean processing speed is 0.2, the saturation point shifts further to the right thereby providing greater flexibility for improving system performance.

The **Efficiency (E)** of a multiprocessor system is another interesting parameter that provides information about the underutilization of processing resource. Effi-

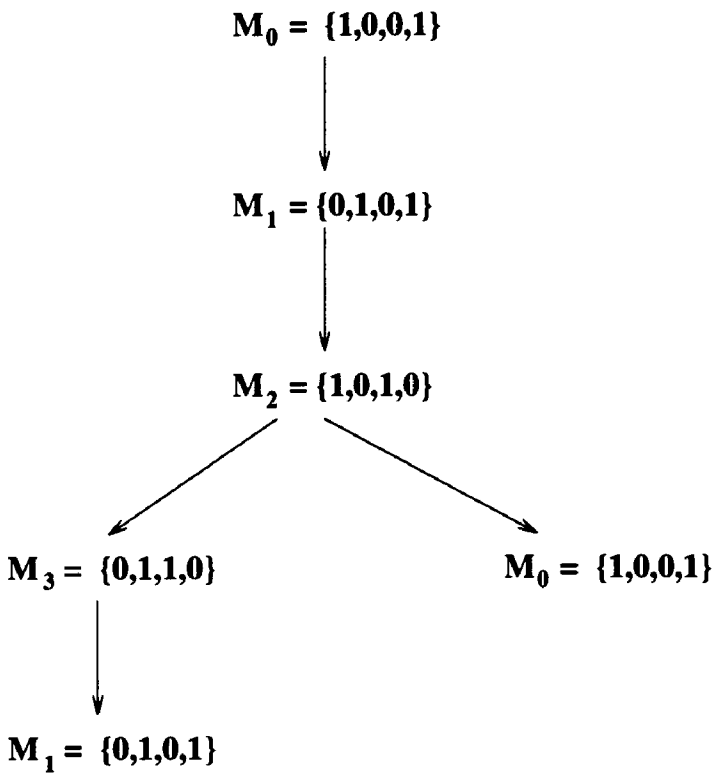


Figure 4.10: Reachability graph of the GSPN model

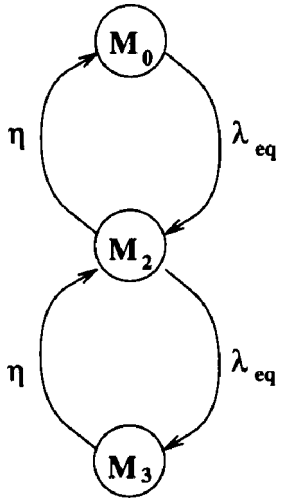


Figure 4.11: Transition rate diagram of the CTMC

ciency is given by the following expression,

$$\begin{aligned} E &= \frac{PU}{m+1} \\ &= \frac{m}{m+1}(1+\sigma\frac{1+m\sigma-m^2\sigma}{1+m\sigma+m^2\sigma^2}) \end{aligned}$$

Figure 4.13 indicates that processor resource utilization characteristic improves as the ratio of L2 and L1 mean processor speeds is decreased. The efficiency curve for $\sigma = 0.2$ not only has a higher peak compared to the other two, average capacity utilization is also much better in this case.

The **Throughput** of the system is an interesting parameter that indicates the average rate of completing computation in a simulation cycle. Note that the throughput of this system is given by the rate at which the L1-processor completes servicing. The states in which L1-processor is active is given by markings that contain atleast one token in place p_2 in the PN diagram. These states are M_2 and M_3. Therefore

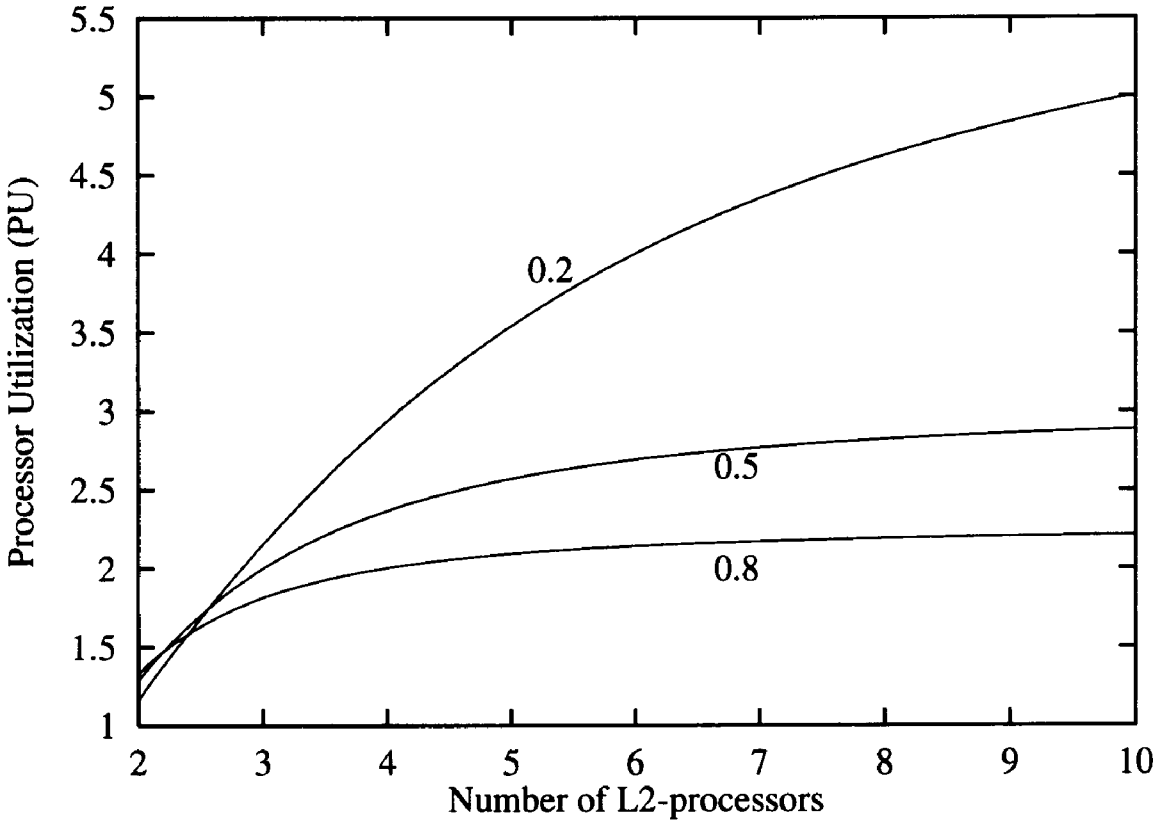


Figure 4.12: PU versus number of L2-processors

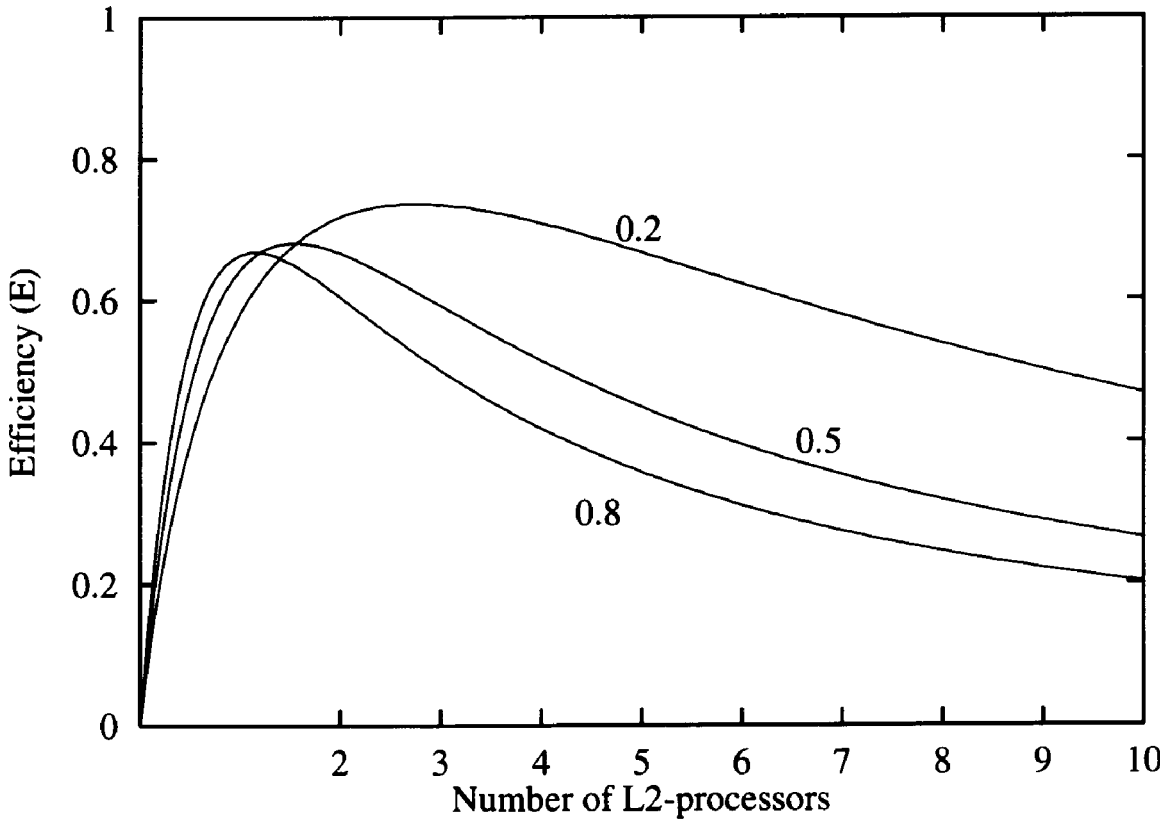


Figure 4.13: Efficiency versus number of L2-processors

the computation throughput is given as,

$$\begin{aligned} T_{comp} &= \sum Prob[place\ p_2\ containsatoken]\eta \\ &= (1-\pi_0^{(c)})\eta \\ &= \frac{m\lambda/\eta + m^2\lambda^2/\eta^2}{1 + m\lambda/\eta + m^2\lambda^2/\eta^2}\eta \\ &= \frac{x^2y + xy^2}{x^2 + y^2 + xy}, \end{aligned}$$

where $x = m\lambda$ and $y = \eta$. Note that the function is symmetric with respect to x and y. Also, increasing x unboundedly makes the function dependent on y and vice versa. Mathematically, it can be shown that the system throughput increases unboundedly only when $x = y$. In all other cases the value of the function is determined by the minimum of x and y. This implies, to maximize throughput $m\lambda$ has to equal η.

4.6 Summary

A new *logic cone* based parallel simulation algorithm was presented in this section. The algorithm performs computation by distributing the workload. An architecture that realizes the computation engine was proposed. It exploits inherent low-level parallelism (*pipelined data-path*) in the algorithm. A novel technique of evaluating the performance of the proposed architecture was introduced.

Chapter 5

Architectural Considerations

5.1 Introduction

Parallelization of synchronized iterative algorithms has produced unexplained and often disappointing results [43, 5]. For a system of p processors, the ideal speedup is p. However, several researchers [44–46] have reported that this speedup diminishes as more processors are used. Clearly there is a point of diminishing returns. In this work, we propose a synchronization architecture based on two dimensional NEWS network with the objective of analyzing the slowdown due to synchronization of router processing elements. Our communication strategy is structured such that for communication intensive jobs, the cost is not of runaway type. We also propose a global synchronization strategy for proper sequencing of events by managing simulation time globally. Performance of the entire simulation system is analyzed using statistical methods based on Markovian theory. Such analysis results in optimized architecture design. Finally in the last section, a load balancing algorithm is presented. This algorithm is particularly useful for allocating computation load between a set of homogeneous processors that do not have a communication bottleneck, as is the case

with our tree of computation processors discussed in the previous Chapter.

5.2 Router Network for Signal Synchronization

When simulation is distributed within multiple processors, it becomes imperative that special efforts are to be made to synchronize simulation clock across the entire set of processors. The only way to achieve this without deadlock is to force all processors into global synchronization by passing appropriate messages. This message passing for simulation clock synchronization is analogous to an all-to-all broadcast problem. The dilemma with this type of high bandwidth communication requirement is that easily implementable (single-chip) and scalable solutions often exhibit poor performance. For example by using a cross-bar switch, one can optimize the communication latency but 90% of the silicon real-estate is wasted in channel footprints. A bus, on the other hand, has very scalable and implementable interconnection structure but provides poor performance. In this section we propose a two dimensional mesh (also called NEWS network) as the inter-cluster communication structure. The mesh is particularly rich in physical connections between router processing elements (RPE). Also, it can be shown that for an n processor system organized as a mesh of trees, the layout complexity is $O(nlog^2n)$ [47]. Note that in our multiprocessor architecture, each RPE in the synchronization plane is connected to its native processor-cluster, which is a a tree of heterogeneous computing elements. RPE is the fundamental component of the router network and can be specially designed to distribute messages along the physical channels or links. Each RPE's native processor-cluster spawns messages and also receives messages from other clusters. The RPE also participates in message propagation. Figure 5.1 illustrates the message distribution paths for a

RPE.

Communication pattern for event synchronization in discrete event simulation is somewhat different from that of other applications. In a simulation cycle, messages are spawned *asynchronously* (separated discrete time points) and then distributed out synchronously using time-division multiplexing. Individual clusters broadcast messages only after their respective computation requirement is completed. The computational load assigned to each cluster is balanced statically to the best of ones ability as described Chapter 3. However during actual simulation, the distribution may become randomly biased because events in a particular clock tick may force computation of only affected blocking devices. This load unbalance is the root cause of asynchronous message spawning in different processors of the mesh. In the discussion that follows, we consider a 16 processor mesh connected as a 4×4 array. The RPEs are labeled in a raster scan order as shown in Figure 5.2, such that each RPE is connected to exactly four neighbors along the north, east, west and south directions. Diameter of this NEWS network is 4.

Message propagation is based on synchronous cycles called *nodal* cycles. Each such nodal cycle is divided into 4 dimension cycles, one for each dimension (north, east, west, south) of the NEWS network. During a nodal cycle, messages are moved across each of the 4 dimensions in a sequence as shown in Figure 5.3. A message is delivered to a neighboring RPE within a single nodal cycle, unless it is delayed due to link contention. A link contention occurs when several messages want to move in the same dimension cycle concurrently. The operation of the router network can be divided into five categories: *injection*, *delivery*, *forwarding*, *buffering* and *referral*.

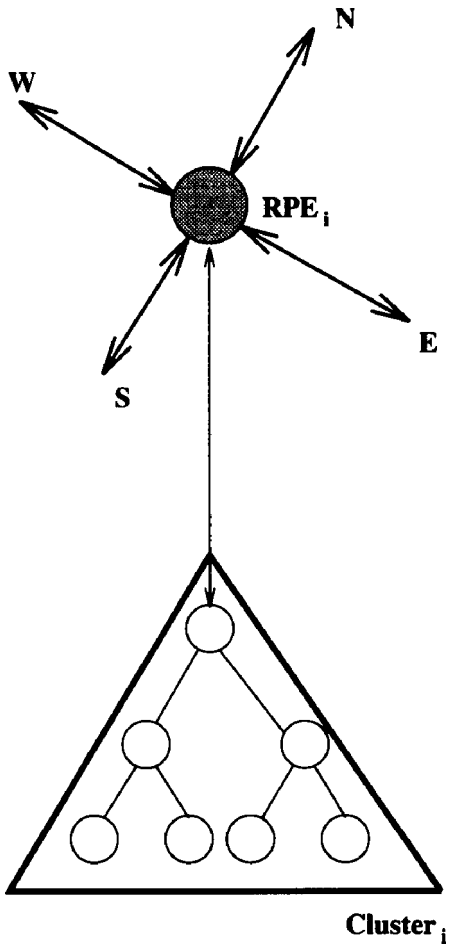


Figure 5.1: Message distribution paths for a RPE

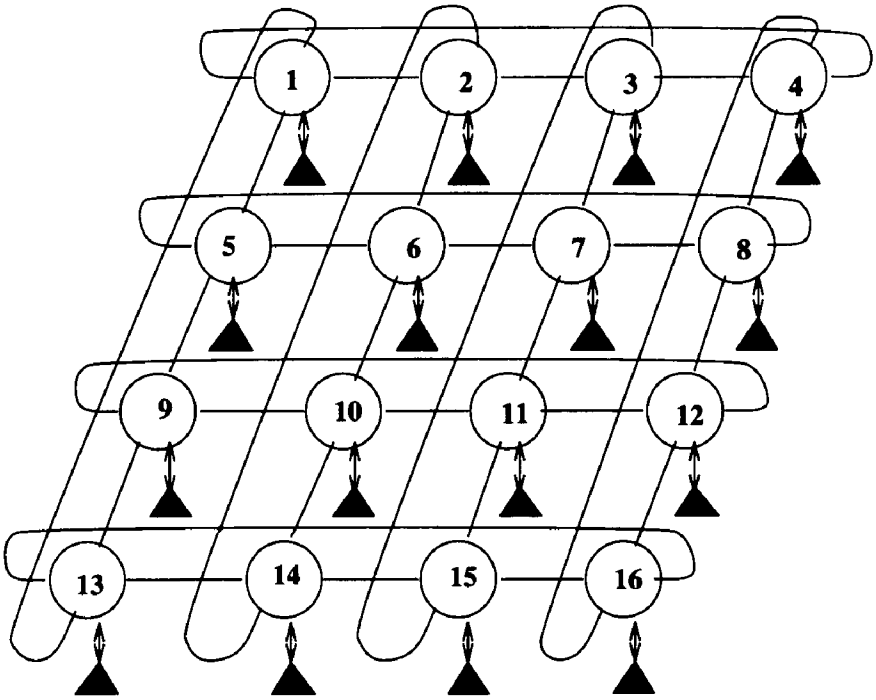


Figure 5.2: Network of RPEs interconnected as a 4 X 4 Mesh

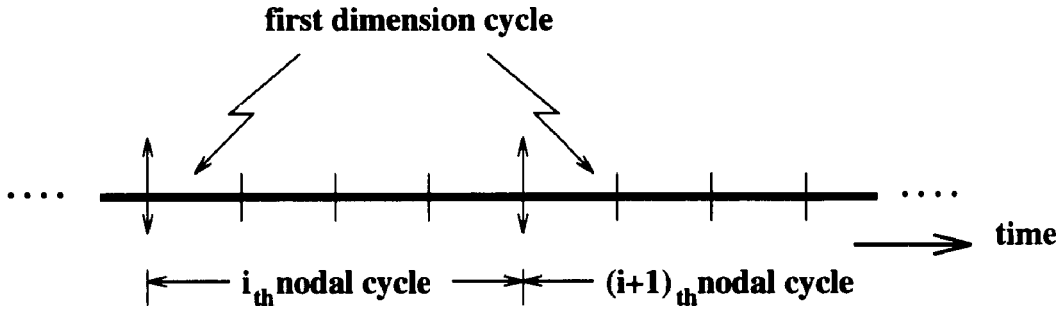


Figure 5.3: Synchronizing mesh communication

Each RPE is connected to a processor-cluster that spawns a message in every simulation cycle. This process is defined as *injection*. The injected message will contain new events that result from evaluation of blocking devices mapped to the native processor cluster. Clearly, this message needs to reach other clusters that are working with blocking devices which will be affected by the events. This message also needs to reach other processors in the network because otherwise these processors will wait indefinitely to receive information from that RPE and this may initiate deadlock. The message distribution overhead varies depending on how many identical messages are generated at a time. RPE accepts no more messages than the size of its buffer.

A message is *buffered* or queued at a router if it cannot be sent in the current nodal cycle due to link contention. The number of buffers at a RPE is 4 (one for each dimension).

In *forwarding*, a message is transmitted from one RPE to another. The destination address of a message is specified relative to the address of the RPE at which it currently resides. If for example, the address is $(+1, +1)$ i.e. one unit south and one unit east of destination, then the message may be moved along *south* or *east* dimension depending upon which dimension is free for this transmission. During a particular dimension cycle, each RPE chooses a message to be sent across the physical link corresponding to that dimension. A router makes this choice by looking at the current relative address (x *and* y co-ordinates). When a message is sent along a particular co-ordinate, its x or y component of the address co-ordinate is reset to maintain address relativity.

Each message is checked for $(0, 0)$ RPE address at the end of a nodal cycle. A message with zero address has arrived at its destination and is *delivered* to the native

processor cluster.

Since the router has limited buffer capacity, a mechanism is needed to deal with a buffer overflow. In a particular dimension cycle, buffer overflow can occur because RPE buffer may be full at the beginning of the cycle and none of the buffered messages can be shifted out in the dimension cycle but a message is received in that dimension cycle. In this situation, RPE intentionally *misroutes* a sitting message to accommodate the incoming message. This process is known as a *referral*. Each referral adds two extra hops to the path —- one hop to misroute and another one to correct it. Thus referral can significantly delay the overall communication time.

5.3 Performance Modeling

Each processor-cluster completes its computation for the simulation cycle and waits for messages to arrive from other clusters. This wasted waiting time is a major drain on the processing power of the machine. Messages entering the router network can generally be forwarded to their destination in one nodal cycle or less. However, sometimes they may get delayed due to link contention. In a nodal cycle, for messages that have non-zero relative address co-ordinates, there are two possible dimension cycles to move out. If there is contention in both these cycle, the message has to wait atleast one full nodal cycle. So there is communication slowdown due to exchange of messages between processors. One may easily see that the asynchronous spawning of messages in the router network makes deterministic performance estimation an extremely difficult task. We have used here Markovian analysis to evaluate the communication delay introduced by the network.

Queuing model of a typical RPE is shown in Figure 5.4. Messages generated by the

native processor-cluster accumulate in the local queue if injection into the network is not possible in the current nodal cycle. Since modeling the entire network is extremely complex, we choose to model a single RPE and extrapolate its performance to all the RPEs. This is a reasonable approximation considering that the network is symmetric and each RPE should behave the same way. The state of the RPE buffer changes in every dimension cycle (discrete time point) and therefore the RPE can be modeled as a DTMC as shown in Figure 5.5. A state of the Markov chain may be represented by a pair $(i,\ j)$, where i represents the dimension cycle within a nodal cycle and j indicates the number of messages in the buffer at the beginning of that cycle. Each cluster spawns a single message in a simulation cycle. We assume that a single copy of this message is distributed in the network by the RPE directly connected to the source cluster. Later, the results of this model will be compared to a scenario where multiple copies of the same message are distributed by the RPE. Given that a message in RPE queue has equal probability of moving out to any of its four neighbors, the probability of being sent out in any one dimension cycle is (1/4). The probability that it is not sent out in that dimension cycle is (3/4). Let $\neg\zeta$ denote the probability of no message moving out. Then the probability that no message moves out of state $(i,\ j)$ is $\neg\zeta_j = (\frac{3}{4})^j$. That means, the probability of a message moving out ζ_j is given as $\zeta_j = 1 - (3/4)^j$. Denote the average queue length of RPE buffer by ϕ. We assume that the network is in steady state, i.e., all neighbors of an RPE exhibit the average behavior. Let $\neg\nu$ denote probability of a RPE not receiving a message from another RPE during a dimension cycle. Then, $\neg\nu = (3/4)^\phi$. The probability of a RPE receiving a message from its neighbor ν, is given as $\nu = 1 - (3/4)^\phi$. Let A_j denote the probability that a message is sent but no message is received in state

$(i,\ j)$ in the current dimension cycle. Then,

$$A_j \ = \ \zeta_j * \neg\nu$$

Similarly let B_j denote the probability that no message is sent but a message is received in state $(i,\ j)$ in the current dimension cycle. Then,

$$B_j \ = \ \neg\zeta_j * \nu$$

Finally, let C_j denote the probability that either [a] no message is received or sent or [b] a message is received and a message is sent in the current dimension cycle in state $(i,\ j)$. Then,

$$C_j \ = \ (1 - A_j - B_j)$$

Non-zero elements of the one-step transitional probability matrix, P_M, may be now calculated as shown below.

Let, $p_{(i,j)(k,l)}$ be the probability of one-step transition from state $(i,\ j)$ to state $(k,\ l)$. When injection is happening at the beginning of the nodal cycle and all buffers are empty, state $(1,\ 0)$, transition is possible to only three other states,

$p_{(1,0)(2,0)} \ = \ u * \neg\nu,$

$p_{(1,0)(2,1)} \ = \ (1-u) * \neg\nu \ + \ u * \nu$ and

$p_{(1,0)(2,2)} \ = \ (1-u) * \nu$

where u is the probability that there is *no* injection.

In the case when injection is happening and not all buffers are empty, states $(1,\ j)\ j = 1, 2, 3, 4$, one has the following non-zero transitional probabilities,

$p_{(1,j)(2,j-1)} \ = \ u * A_j,$

$p_{(1,j)(2,j)} \ = \ u * C_j \ + \ (1-u) * A_j,$

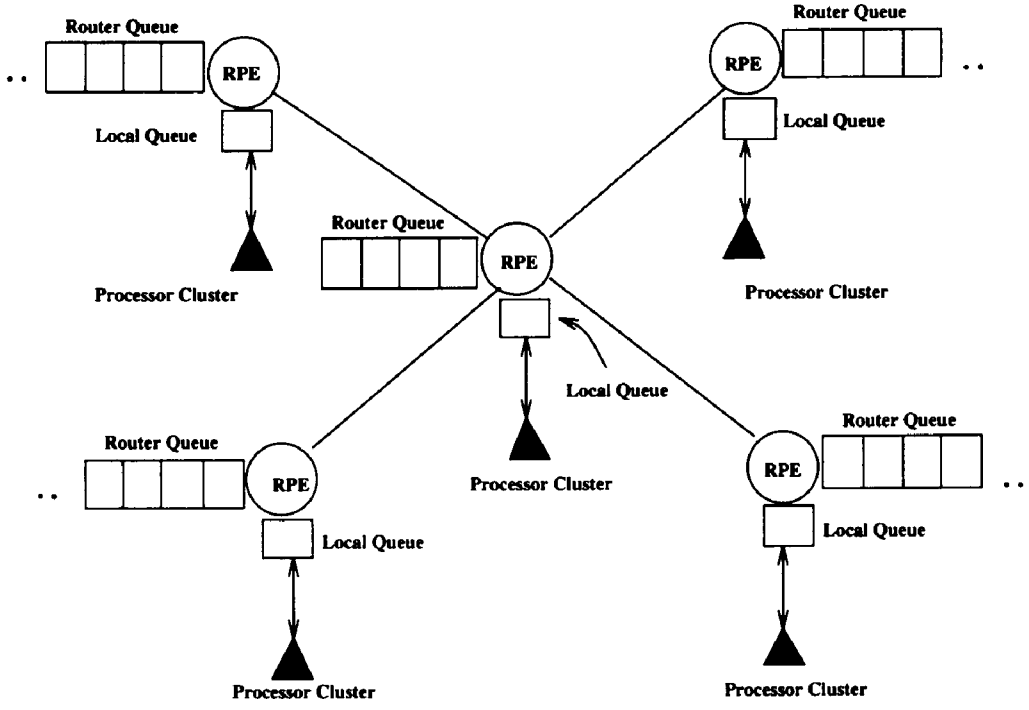


Figure 5.4: Queuing model of an RPE

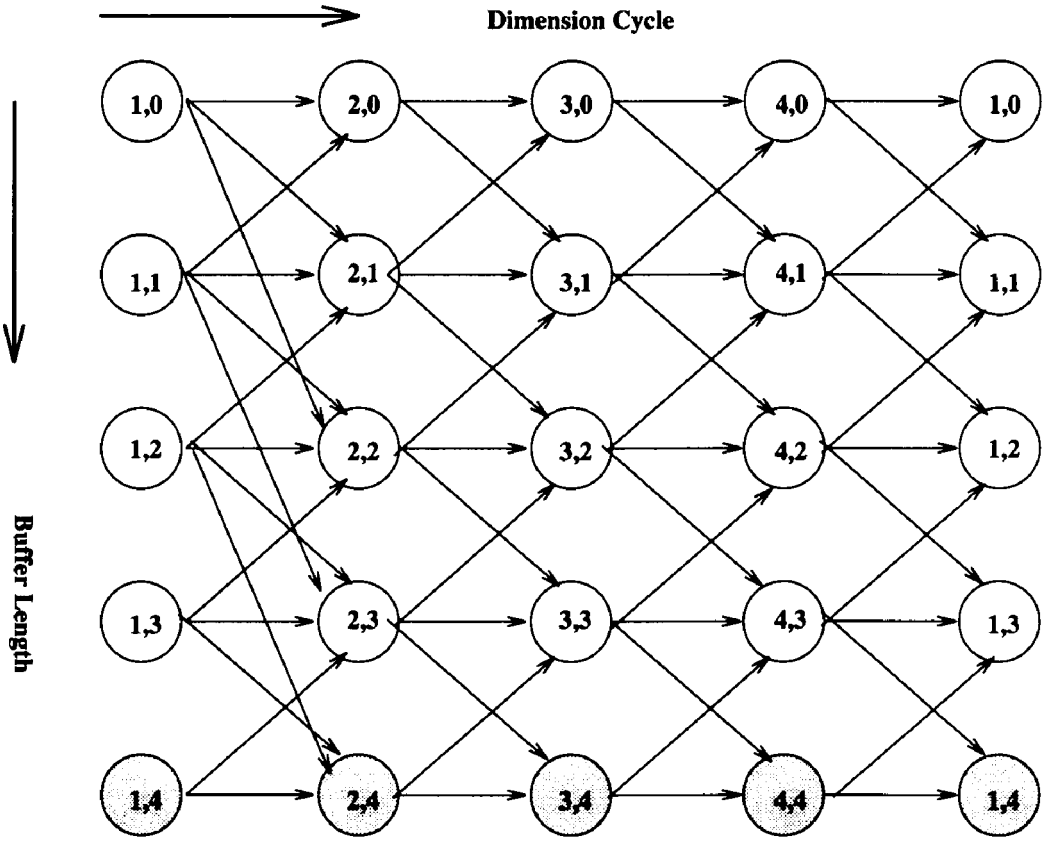


Figure 5.5: Discrete time Markov Chain for the RPE

$p_{(1,j)(2,j+1)} = (1-u) * C_j + u * B_j$ and

$p_{(1,j)(2,j+2)} = (1-u) * B_j$.

In case injection does *not* take place and all buffers are empty, states $(i,\ 0)\ i = 2,3,4$, no message can be sent out but may be received. we have the following non-zero transitional probabilities,

$p_{(i,0)(i+1,0)} = \neg\nu$ and

$p_{(i,0)(i+1,1)} = \nu$.

In case injection is *not* happening and all the buffers are full, states $(i,\ 4)\ i = 2,3,4$, a message is always sent out because each slot of the buffer is meant for a message along a particular direction. we have the following non-zero transitional probabilities,

$p_{(i,4)(i+1,3)} = \neg\nu$ and

$p_{(i,4)(i+1,4)} = \nu$.

In any other state, $(i,\ j)\ i = 2,3,4$ *and* $j = 1,2$ *and* 3, we have the following non-zero transitional probabilities, obtained directly from definitions:

$p_{(i,j)(i+1,j-1)} = A_j$,

$p_{(i,j)(i+1,j)} = C_j$ and

$p_{(i,j)(i+1,j+1)} = B_j$.

Let $\pi_{(i,j)}$ be the steady state probability of being in state $(i,\ j)$. Clearly, $\sum_{(i,\ j)}\pi_{i,j} = 1$. Recall from previous Chapter that we can get the steady probability distribution by solving the following equation,

$$[\pi] = [\pi] \cdot [\mathbf{P_M}],$$

where $\mathbf{P_M}$ is the one-step transition probability matrix and π the probability vector. Since entries in $\mathbf{P_M}$ are functions of ϕ, the average queue length, which itself is a function of π, a fixed point iteration is required to solve the above equation and

obtain the values of individual probabilities.

We now derive expressions for performance measures in terms of the steady state probabilities of the Markov chain. Two of the parameters we focus on are the average queue length at an RPE and the system throughput. Both of these are important measures of the hardware and time complexity of the system. We will also show later that they together, provide one with the "response time", one of the most important measures used to quantify the time complexity of a simulation tick. One may easily see that the system throughput (Λ) may be computed as the number of messages transferred per unit of time at a RPE. It is the sum of the steady state probabilities of the states in which messages can be sent by a router node, weighted by the probability of sending a message in that state. Note that when the buffer is full, in states (i, 4), the probability of sending a message is 1 because a each slot of the buffer is allotted to for a particular dimension cycle. Hence,

$$\Lambda = \sum_{i=1}^{4}\sum_{j=0}^{3}((1-(\frac{3}{4})^j) * \pi_{(i,j)}) + \sum_{i=1}^{4}\pi_{(i,4)}$$

The throughput due to referral, denoted by Λ_{ref}, may be defined as the number of messages referred per unit of time at a RPE. It is the sum of the steady state probabilities of the states in which buffer is completely full, weighted by the probability that the message is referred. Hence,

$$\Lambda_{ref} = \sum_{i=1}^{4}(\frac{3}{4})^4 * \pi_{(i,4)}$$

The probability of referral, P_{ref}, is the probability that a message is referred at a RPE and is given by the following expression:

$$P_{ref} = \frac{\Lambda_{ref}}{\Lambda}$$

The average queue length, ϕ, is the expected number of messages in the buffer at a router and is given by the following expression:

$$\phi = \sum_{i=1}^{4}\sum_{j=0}^{4}(j * \pi_{(i,j)})$$

The average number of hops (Ω) taken by a message to reach all its destination will depend on the symmetric message distribution pattern. If the distribution pattern is such that each RPE forwards a copy to one of its neighbors only (in a ring fashion), the average distance is given by $\frac{\sum_{i=1}^{15} i}{15}$. If on the other hand the distribution pattern is such that each RPE forwards a copy of the message to all four neighbors, the average distance is 2.1. The average number of hops due to referral, Ω_{ref}, is the average number of hops taken by a message due to referral. The average total number of hops, Ω_{tot}, is the sum of hops without any referral plus the hops due to referral, i.e.,

$$\Omega_{tot} = \Omega + \Omega_{ref}$$

For each referral, a message typically takes two extra hops, one hop due to misroute and another to correct it. At every RPE, a message has a risk of being referred with probability P_{ref}. The probability of referral is independent at each RPE. Hence, the average number of hops due to referral is $\Omega_{tot} * P_{ref} * 2$. Thus,

$$\begin{aligned}\Omega_{tot} &= \Omega + \Omega_{tot} * P_{ref} * 2 \\ &= \frac{\Omega}{1 - 2 \cdot P_{ref}}.\end{aligned}$$

Now we calculate the average time spent by a message in the local queue. In every simulation cycle, a message is spawned by processor-cluster native to each RPE. We assume that the loading of cluster is such that it generates the message somewhere within the first nodal cycle. Note that, if a message is not generated in the beginning

of a nodal cycle, it has to wait atleast till the beginning of the next nodal cycle. Assuming a dimension cycle to be of *unit time*, average waiting time in a nodal cycle is 2. Given the probability of no injection in the current nodal cycle, u, probability that the message will enter the network at the beginning of the next nodal cycle is $u(1-u)$, $u^2(1-u)$ the cycle following that and so on. So, expected value of waiting time in the local queue is given by the infinite series,

$$\begin{aligned} T_{locQ} &= 2u(1-u) + 4u^2(1-u) + 6u^3(1-u) + 8u^4(1-u) + \ ... \\ &= 2u(1-u) \cdot [1 + 2u + 3u^2 + 4u^3 + \ ...] \\ &= 2(1-u) \cdot \frac{dS}{dq} \end{aligned}$$

where, $S = (u + u^2 + u^3 + \ ...)$.

Since, $u < 1$, this infinite sum collapses into,

$$T_{locQ} = \frac{2u}{(1-u)}.$$

Finally, the response time, R, is defined as the total time spent by a message in the router network —- from submission to the local queue at the source RPE till arrival at the destination RPE. It is the sum of wait time in local queue of the RPE and the time spent in the network after injection. According to Little's Law, the average time spent by a message at a RPE is $\frac{\phi}{\Lambda}$. Thus the average time spent by the message in the network is $\Omega_{tot} * \frac{\phi}{\Lambda}$. Added with the time spent in the local queue one gets response time,

$$R = \Omega_{tot} * \frac{\phi}{\Lambda} + \frac{2u}{1-u}$$

.

5.4 Numerical Results and Analysis

The analytic model presented in the previous section was solved to gather performance measures of the signal synchronization network. Since derivation of steady state probability distribution requires the knowledge of ϕ, which is an unknown, we used an iterative procedure called *fixed point iteration*. Fixed point iteration is an accurate and cost-effective technique to analyze queuing networks when the exact analysis is numerically intractable. Fixed point iteration has been previously used in other applications [48–51].

We used fixed point iteration to solve our model in the following manner:

Step-1: Select an initial value of ϕ in the range 0-4.

Step-2: Compute elements of one-step TPM $\mathbf{P_M}$.

Step-3: Solve for steady-state probability distribution.

Step-4: Using these values find the new value of ϕ.

Step-5: check for new value of ϕ to be within 1% of old value.

Step-6: repeat (back to step-2) until step-5 is true.

Even with different initial values of ϕ, the solution converged to identical values in just 4-6 iterations. This clearly indicates that a unique solution exists to the fixed point iteration problem and the convergence is rapid. Performance data was collected for different injection probabilities, $1-u$, ranging from 0.1 to 0.9. Note that $1-u$ can also be interpreted as the arrival probability of the message generated in the RPE.

Table 5.1 below shows the performance measures (throughput, average queue length, response time, average hop length and probability of referral) for various values of injection probability. Note that, $1-u$, in the analysis above represents the injection probability.

injection prob.	throughput	avg. queue length	prob. of referral	hops due to referral	avg. num. of hops
0.1	0.44	1.71	0.036	0.358	8.07
0.2	0.52	1.99	0.053	0.531	8.10
0.3	0.58	2.17	0.066	0.665	8.13
0.4	0.64	2.39	0.083	0.835	8.16
0.5	0.69	2.56	0.098	0.980	8.19
0.6	0.74	2.71	0.111	1.110	8.22
0.7	0.78	2.85	0.124	1.240	8.24
0.8	0.81	2.96	0.135	1.350	8.27
0.9	0.84	3.07	0.146	1.460	8.29

Figure 5.6: Performance measures for different injection probabilities

As expected, throughput increases as we increase the injection probability. But this increase is not linear. Changing the injection probability from 0.1 to 0.2 results in small throughput increase of 0.012. However, the throughput increases by 0.31 when the injection probability is increased from 0.8 to 0.9. This is because injection, in general, has a positive effect of improving the overall throughput.

We observe that the average queue length increases as we increase the message injection probability. This is because as the injection probability $(1 - u)$ is increased, expected number of messages in the RPE queue also increases.

We observe that the probability of a message being referred increases as we increase the injection probability. As noted earlier, the average queue size increases as we increase the injection probability. This increases the likelyhood of the RPE queue being full resulting in referral.

Results demonstrate that the average number of hops due to referral increase as the injection probability increases. This is because the probability of referral increases as we increase the arrival probability, thereby, increasing the average number of hops due to referral.

Again, we observe that the average total number of hops taken by a message increases as we increase the arrival probability. This is because the average number of hops due to referral increases as we increase the arrival probability.

Figure 5.7 shows that unlike all other parameters, response time does not continuously increase or decrease in the injection probability domain. It first decreases as arrival probability is increased from 0.1 to 0.6. However, from 0.6 to 0.9, it climbs back again. This behavior is mainly attributed to the non-linearity of system throughput. When the probability of message arrival is very low (injection probability close to 0), injection is not going to happen in the first dimension cycle of the current nodal cycle. Note that in a nodal cycle, injection can only happen in the first dimension cycle. So, the native message has to wait in the local queue atleast till the beginning of the next nodal cycle. This obviously increases the response time. On the other hand, if injection is almost certainly going to happen in the current nodal cycle (injection probability close to 1.0), it is possible that the native message did not make it to the RPE queue because it is full. This results in referral and the native message has to spend time in the local queue. The response time characteristic curve also indicates that the response time degradation due to injection is smaller than due to no-injection.

Single message distribution pattern, as shown in Figure 5.8, was used for the analysis above. We also experimented with a symmetric multiple (4-way) message

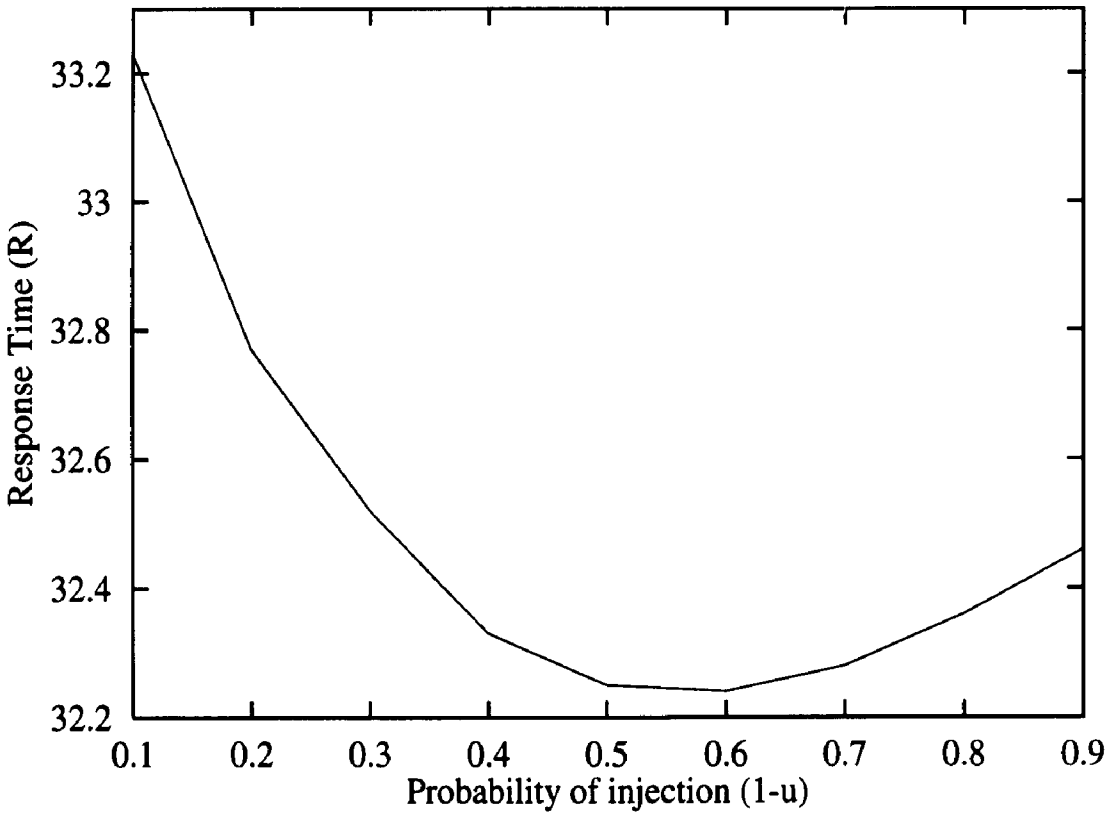


Figure 5.7: Response time versus arrival probability

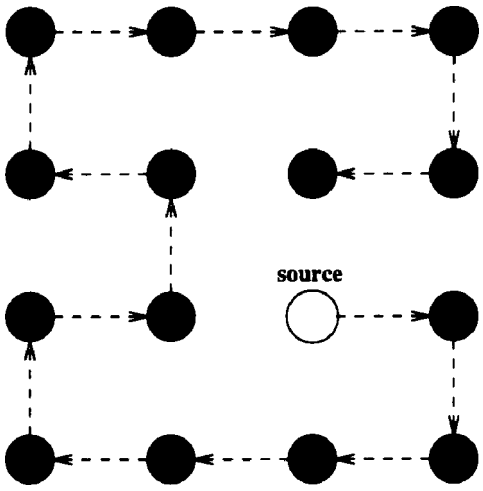


Figure 5.8: Single way message distribution pattern

distribution pattern shown in Figure 5.9. Each processor cluster still spawns a message, however, the native RPE now distributes messages in all 4 directions as opposed to one. The distribution pattern is such that message propagation can proceed symmetrically in all four directions. Intuitively, the response time should reduce four fold. However this is not the case. For an injection probability of 0.5, the response time for the 4-way distribution is 10.86 time units as opposed to 32.25 units for the 1-way distribution pattern. The extra time is due to distribution redundancy and more referrals. With 3 additional messages originating at each RPE, there will be more queue congestion and therefore more referrals. The 4-way distribution pattern suggested here has some redundancy. Some of the RPEs in the synchronization network receive the same message more than once. A more optimized yet symmetric pattern distribution strategy is beyond the scope of this work.

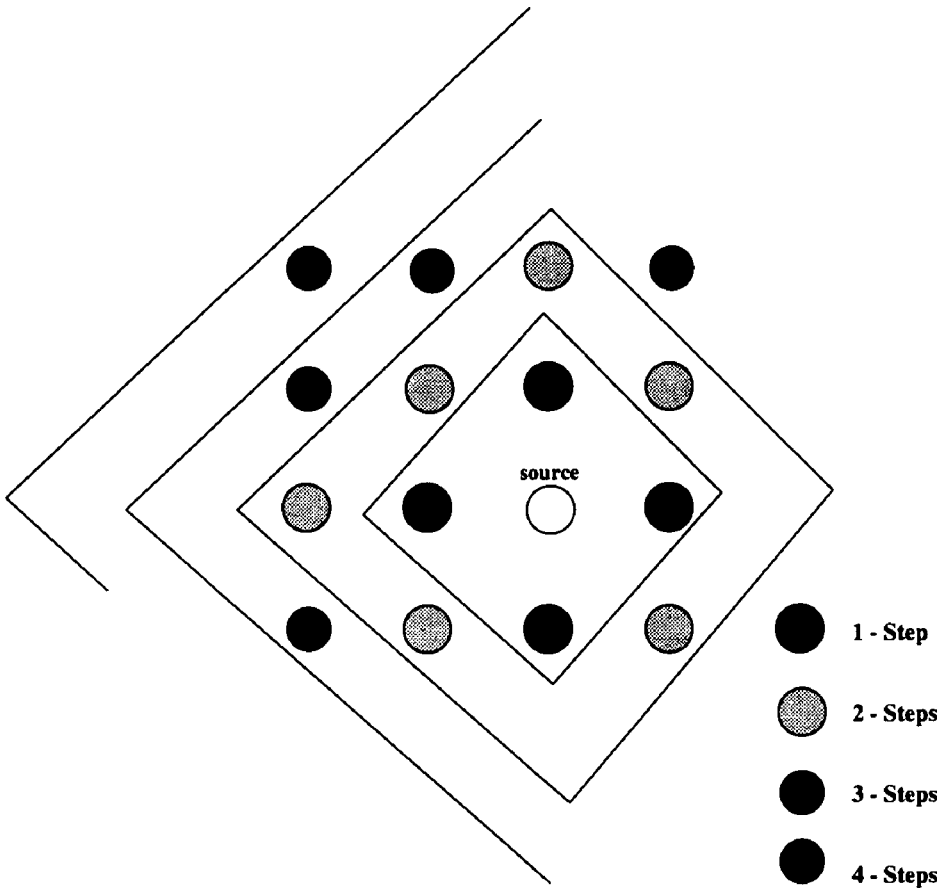


Figure 5.9: 4-way message distribution pattern

5.5 Global Time Manager

In the previous Chapter, we presented a heterogeneous group of architecturally specialized processors, interconnected as a tree, that perform the computation part of the parallel simulation algorithm. Total computation load of simulation is shared by a number (actually 16 here) of such trees (also called *processor–clusters*). Processor-clusters synchronize their new signal status through a network of RPEs organized as a two dimensional NEWS network, presented in the previous section. The computation and communication load of a simulation cycle has to be succeeded by time-synchronization to allow correct sequencing of events, *without deadlock*. We propose a Global Processor Element (GPE) that manages simulation time. It also has the responsibility of doing miscellaneous house-keeping tasks like scheduling new primary input or driver values, monitoring transitions on critical signals like primary outputs and interrupting host for more data. The GPE should be connected to all the RPEs through dedicated buses. Since both the RPE and the GPE do not try to access the bus at the same time, there is no contention for the bus resource connecting RPEs to the GPE. The layout complexity, due to channel footprint does not change, because a linear term is added to the complexity expression $O(nlog^2n + K)$, where n is the total number of processors in the multiprocessor system and K indicates the number of RPEs ($K \ll n$). Synchronization is achieved by using a variable to count the number of *done* messages received along the RPE-GPE channels. When the count variable reaches a threshold (16 here), the GPE moves to the next simulation cycle, unless there are no more simulation cycles to process. The count variable is a critical resource in this approach. Simultaneous requests to increase the count variable are handled sequentially. Each time a request is considered for service, the count-variable

is locked till the servicing is completed. This type of sequential processing imposes a limit on how many RPEs can be accommodated in the network. Obviously, for the parallel simulation solution to be efficient, time synchronization complexity has to be several orders of magnitude lighter than computation plus communication complexity. Control-flow in the GPE proceeds as follows.

Step-1: GPE broadcasts *start* message and resets *count*.

Step-2: PI or driver-buffer is re-filled, if empty.

Step-3: Unload output-buffer (contains signal transitions), if full.

Step-4: Wait for *done* messages.

Step-5: Respond to request for count-increment service.

Step-6: If *count* not equal to threshold, go back to Step-4.

Step-7: If there is more to simulate, go back to Step-1.

Note that steps 2 and 3, providing user interface to the simulation, are optional and do not necessarily happen in every cycle. Also, these steps start with the GPE interrupting the host for service. Our studies show that the average time to walk through steps 1-3 constitute less that 15% of the workload. If we neglect this overhead, the GPE can be modeled as an open queuing network with requests for service arriving along K channels. We assume a poisson arrival rate δ of the "done" messages along each channel since the arrivals along individual channels are independent. Note that the network is open because the next cycle cannot begin until all K requests have been served. Let the buffer be of size S, where $1 \geq S \leq K-1$. The buffer needs to be at most $K-1$ because at most K requests can pile up (one being served) in the buffer. Let ψ denote the exponential rate of servicing. Then we have a M/M/1/S queuing system, as shown in Figure 5.10, that models the concurrent processing of

the GPE and K RPEs. It can be shown [52] that such a queuing system has $S+2$ steady states and the steady state probability distribution of each state is given as,

$$\pi_n = \pi_0(\frac{K\delta}{\psi})^n, \ \forall n = [0, S+1].$$

Also since sum of all the steady state probabilities is 1, π_0 can be calculated as follows.

$$\pi_0 = \frac{1}{1 + K\kappa + K^2\kappa^2 + \ ... \ + K^{S+1}\kappa^{S+1}},$$

where $\kappa = \frac{\delta}{\psi}$.

Throughput of the system is an interesting parameter that indicates the average rate of completing a simulation cycle. Note that with the exception of the initial state (buffer empty and GPE waiting for request to arrive), in all the other states GPE is busy serving a request. Therefore the system throughput, T_{system}, in this context is given as,

$$\begin{aligned} T_{system} &= \sum Prob[in\ a\ state\ other\ than\ initial\ state]\psi \\ &= (1 - \pi_0)\psi \\ &= (1 - \frac{1}{1 + K\kappa + K^2\kappa^2 + \ ... \ + K^K\kappa^K})\psi, \ Assuming\ S = K - 1. \\ &= \frac{x/y + (x/y)^2 + \ ... \ + (x/y)^K}{1 + x/y + (x/y)^2 + \ ... \ + (x/y)^K} y, \end{aligned}$$

where $x = K\delta$ and $y = \psi$. Note that the function is symmetric with respect to x and y. Also, increasing x unboundedly makes the function dependent on y and vice versa. Mathematically, it can be shown that the system throughput increases unboundedly only when $x = y$. That means $K\delta$ has to be equal to ψ. The design implication of this analysis is that for a $\frac{\delta}{\psi}$, the value of K for maximum throughput condition is fixed.

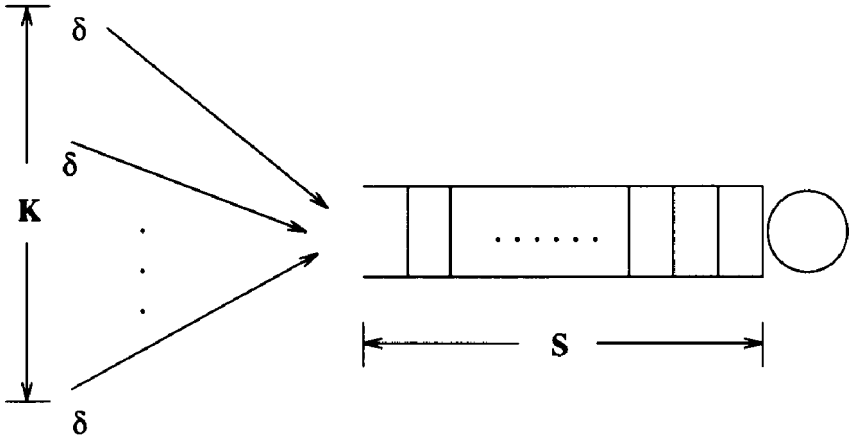


Figure 5.10: Queuing model of the global processor element

In the previous Chapter, we derived the optimum architectural conditions for each processor cluster. For the two dimensional funneled pipeline, throughput (λ) is maximum when $\mu_2/\alpha = \mu_1$. Let t stage-2 processors of the pipeline working in parallel, each with an average speed of μ, result in the optimum behavior. Then, the average speed of the first stage of the pipeline, μ_1, has to be $t\mu/\alpha$. Using this relationship in the throughput expression for the two dimensional funneled pipeline we have,

$$\begin{aligned} \lambda_{max} &= \frac{2}{3}\mu_1 \\ &= \frac{2}{3} \cdot \frac{t\mu}{\alpha} \end{aligned}$$

Also, throughput of each processor cluster, T_{comp}, is maximum when $m\lambda = \eta$. By substituting this relation into the throughput expression, maximum throughput of a processor cluster is obtained as,

$$T_{comp_{max}} = \frac{2}{3} m \lambda_{max}$$

$$= \frac{2}{3}m(\frac{2}{3} \cdot \frac{t\mu}{\alpha})$$
$$= \frac{4mt\mu}{9\alpha}$$

In the previous section, we derived the maximum throughput condition per simulation cycle for the entire system as $\psi = K\delta$. This relation results in the maximum simulation throughput given as,

$$T_{system} = \frac{K^2}{K+1}\delta$$

where δ represents the arrival rate of jobs at the GPE. This rate is determined by the average computation time per cycle and the synchronization time. We computed the average computational throughput per simulation cycle. Let $R(K)$ be the average response time due to synchronization for a K processor NEWS network. Then we get the expression of δ as,

$$\delta = \frac{1}{1/T_{comp_{max}} + R(K)}$$

Using this value of δ to calculate the system throughput we get,

$$T_{system} = \frac{K^2/K+1}{9\alpha/4mt\mu + R(K)}$$

This expression has some interesting characteristics. For small values of K, m *and* t —- system throughput increases almost linearly. However, as K becomes fairly large (greater than 16), $R(K)$ grows faster than $\frac{K^2}{K+1}$ thereby reducing the throughput. This explains why massive parallelism is not good for logic simulation. Intuitively, cost due to synchronization starts hurting the gain due to parallelization. Let λ_0 be the simulation throughput per cycle on a base machine (sparc station for example). Then **speedup (S)** is defined as the ratio of the throughput of our multiprocessor

system to the throughput of the base machine. Speedup is given as,

$$\begin{aligned} S &= \frac{K^2/(K+1)}{3\lambda_0/2m\lambda_{max} + \lambda_0 R(K)} \\ &= \frac{K^2/(K+1)}{(3/2mH) + \lambda_0 R(K)}, \end{aligned}$$

where H is the throughput gain due to hardware specialization and is given as,

$$H = \lambda/\lambda_0.$$

For $K = 16$, response-time $R(K)$ for synchronization was calculated to be 33 dimension cycles. Assuming each dimension cycle to be of one micro-second duration, the response time is 33 micro-seconds. Also assuming typical vales of m, H and λ_0 as 2, 250 and 500 events/second respectively, we get a speedup of approximately 1000 times. This clearly indicates that our accelerated simulation solution can shorten the design verification cycle tremendously. However, one should note that, since $R(K)$ tends to grow exponentially with increase in K, scaling the architecture will not result in a better solution.

The entire architecture is shown in Figure 5.11.

5.6 Intra Cluster Partitioning Algorithm

In the previous Chapter, it was discussed that in order to extract inherent parallelism in the problem one has to partition the signal and device evaluation workload. This way both these computations can proceed concurrently. Since the computational complexity of device evaluation is several orders of magnitude higher than its signal evaluation counterpart and tends to dominate the computation time, overall throughput can be improved by distributing this workload. A tree interconnection

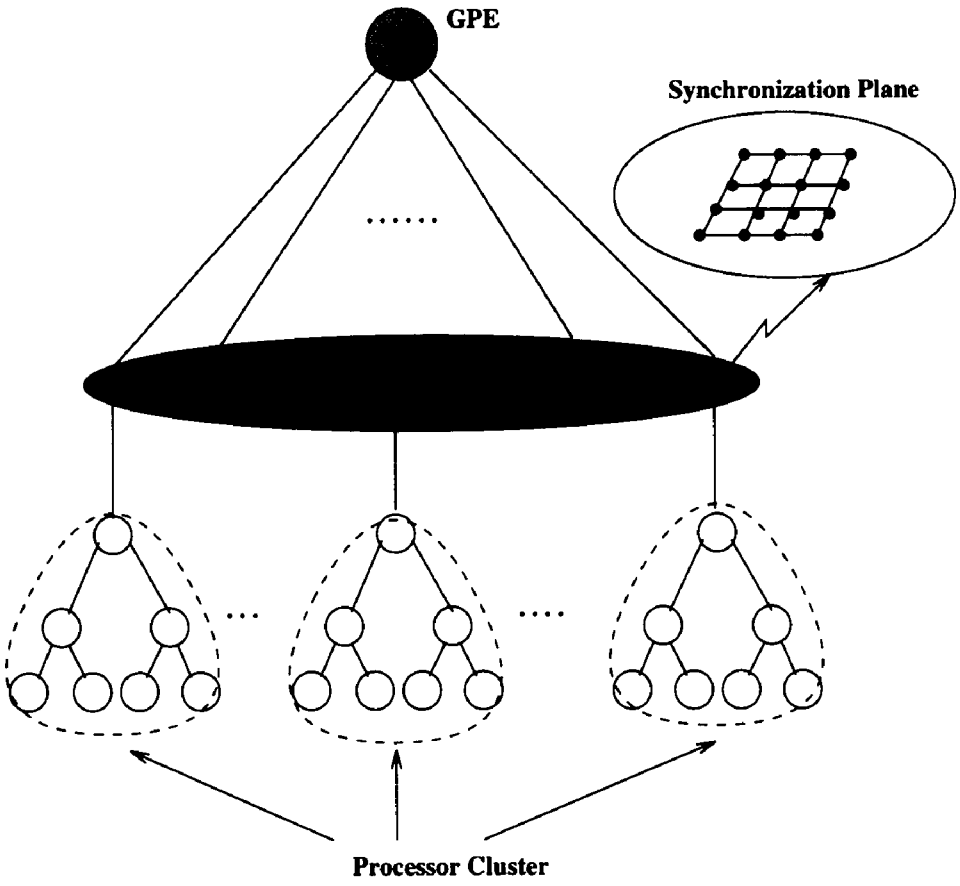


Figure 5.11: Complete architecture of the simulation engine

structure with the root being a signal-PE and all the leaf PEs as device processors fits this structure. However, one has to note that complexity distribution strategy will work only if the device evaluation workload is well balanced among all the device evaluation PEs in any simulation cycle. In other words, in each simulation cycle, all the leaf PEs of the cluster must complete their work at almost the same time. In this section we discuss an algorithm for such load balancing.

Let Q_i be the fanin signal requirement for the i_{th} processor cluster. Q_i contains the set of all the signals that feed blocking-devices assigned to i_{th} cluster. Such assignment of blocking-devices is done using the algorithm discussed in Chapter 3. Then, there are $2^{|Q_i|}-1$ combination of distinct *active* signal patterns. A signal pattern is active if it drives the next tick device evaluations for the cluster. We assume that each signal-pattern has equal probability of becoming active. Let m_j be the number of inputs for the j_{th} blocking-device in the i_{th} cluster. Then there are $2^{|Q_i|-m_j}$ signal patterns that when active does not trigger the computation of blocking-device j. So, the probability that the j_{th} blocking-device is *not* excited is $\frac{2^{|Q_i|-m_j}}{2^{|Q_i|}-1}$. Then, the activation index or activation probability $I(i,\ j)$ is given by,

$$I(i,\ j)\ =\ 1-\frac{2^{|Q_i|-m_j}}{2^{|Q_i|}-1}.$$

The workload per simulation cycle can therefore be matched by assigning the blocking devices with heavier activation index first in a round-robin fashion among all the leaf PEs. The algorithm to balance the device computation load is as follows:

Step-1: Sort blocking devices in a cluster by decreasing order of activation index.

Step-2: Assign them in a round-robin fashion with the more active ones first.

5.7 Summary

In this section we proposed an interconnection structure to handle asynchronous communication due to signal synchronization. The communication strategy is well bounded both for computation and communication intensive jobs. Architectural consideration for supervising the proper sequencing of events was also considered. We used statistical techniques based on Markovian theory to analyze the performance of our simulation system. The performance analysis measure was also used to motivate optimization of architectural overhead. Finally, an intra cluster load balancing strategy was introduced to optimize the workload distribution of computation load.

Chapter 6

Conclusions

6.1 Summary

With the increase in size and complexity of digital circuits, it has become extremely important to verify correct operation of a digital circuit. Such verification can take prohibitive amount of time and therefore one has to come up with either better heuristics for uniprocessor solutions or use the tremendous computing power of multiprocessors. General purpose multiprocessors are not particularly well suited for simulation. Sophisticated caching mechanism is required to meet the high bandwidth memory requirement during simulation. The alternative is to design special purpose multiprocessors using custom or of the shelf microprocessor components that cater to the specific computing and memory requirements of simulation. Design and analysis of efficient parallel algorithms and architecture were the focus of this thesis. All architectural considerations were qualified using exhaustive analysis based on Markovian theory.

In Chapter 2 various parallel processing techniques for VLSI simulation were categorized with their individual merits. Important decisions for multiprocessor design

were examined. Both current and previous work in acceleration techniques for VLSI simulation were also reported in this Chapter. Finally, input data partitioning strategy for our multiprocessor based simulation was outlined.

In Chapter 3 we proposed a novel assignment algorithm for mapping input data graph onto processor graph. Our algorithm tries to minimize the penalty due to mis-assignment. The algorithm is efficient and yields superior simulation runtime characteristic. It is fairly general and can be applied to other types of discrete event simulation.

In Chapter 4 a parallel simulation algorithm based on distributed event management was outlined. The algorithm was verified for correctness using Petri net theory. We also presented a specialized multiprocessor system realizing the proposed algorithm. Stochastic Petri net theory was used to analyze the performance of the system and examine various design tradeoffs.

In Chapter 5 we proposed an interconnection system in the form of two dimensional NEWS network to handle asynchronous communication due to signal synchronization. Architectural details for guarding event sequencing were also examined in this Chapter. All architectural considerations were analyzed using statistical methods based on Markovian theory. Finally, an intra-cluster load balancing algorithm was introduced to equitably distribute computation workload among processors.

6.2 Future Research

As a part of future research, it would be interesting to see whether any of the techniques presented in this work can be applied to other CAD problems. For example, one can develop an accelerated differential fault simulation [53] engine using the accel-

eration techniques proposed in this dissertation. Like gate level functional verification, fault simulation is timing independent. Further, in differential fault simulation, difference between the faulty machine and the good machine is stored only in sequential (blocking) devices and propagated forward based on difference at these critical sites. Storing the difference would mean additional overhead but this is minimal compared to concurrent fault simulation, where the difference list is propagated through all devices. Consider a fault in a particular cone. The faulty machine is evaluated by overlaying the effect of fault on the cone definition and then simulating. The difference at an intermediate blocking device is not propagated, if the results concur with the good circuit simulation. A fault is declared detected, if the difference reaches an observable point.

Another interesting extension to our work would be to implement accelerated multiple delay simulation. Each cone could be treated as a supergate with pin-to-pin delays. The main challenge would be to devise accurate pin-to-pin delay distribution methods for these supergates. Depending upon which set of blocking device inputs carry events, time-stamp of the next event on the blocking device output will be different. The next active simulation tick will not necessarily be the next tick. Unequal pin-to-pin supergate delays result in uneven event distribution. For stretches of ticks there may not be any event at all, where as for some other ticks there will be event congestion. Parallelism can be better utilized for such non-uniform event distribution. This is because, the GPE algorithm can be easily modified to skip ticks without any events and event congestion is better handled by hardware specialization. Further investigation is needed to develop parallel algorithms in other areas of electronic CAD like test-generation and logic synthesis.

Multiprocessor design can be fairly complex. Synthesizing such designs is an iterative procedure. High level modeling and analysis tools are required to come up with the most architecturally efficient design. In this work we have introduced a powerful technique in the form of generalized stochastic Petri nets to do such analysis. However, GSPN theory needs further development in certain areas. Size of the state space in the reachability graph derived from the Petri net tends to grow exponentially. Markov models for such large reachability graphs demand excessive amount of computing resource for a numerical solution. Further investigation is needed to devise ways of aggregating states of the Markov model or reachability graph to come up with a numerically tractable solution. Another alternative is to decompose the problem hierarchically according to some time-scale. Workload detail increases as the time scale resolution is increased. So a given problem can be broken into many subnets, each representing a portion of the system. This divide and conquer strategy works well in many applications, but when computing the overall system performance by aggregating the parameters at different levels, this approach may be prone to errors. Further investigation is needed to measure the magnitude of such error due to aggregation.

Bibliography

[1] K. T. Tam, "Parallel processing for cad applications," *IEEE Design and Test*, pp. 13–17, Oct. 1987.

[2] J. Sargent and P. Banerjee, "A parallel row based algorithm for standard cell placement with integrated error control," in *Proc. of 26th ACM/IEEE Design Automation Conference*, June 1989.

[3] C. B. Erickson, *Design and Evaluation of Hierarchical Bus Multiprocessors.* Ph.D. dissertation, University of Michigan, 1991.

[4] A. E. Ruehli and G. S. Ditlow, "Circuit analysis, logic simulation and design verification for vlsi," *Proceedings of IEEE*, vol. 71, pp. 34–48, Jan. 1983.

[5] L. Soule and T. Blank, "Statistics for parallelism and abstraction level in digital simulation," in *Proc. of ACM/IEEE Design Automation Conf.*, pp. 588–591, June 1987.

[6] M. L. Baily and L. Snyder, "An emperical study of on-chip parallelism," in *Proc. of 25th ACM/IEEE Design Automation Conference*, June 1988.

[7] S. A. Kravitz, "Massively parallel switch level simulation: A feasibility study," Tech. Rep. CMUCAD-89-45, CMU, 1989.

[8] M. A. Breuer and A. D. Friedmann, *Diagnosis and Reliable Design of Digital Systems.* Rockville, Md.: Computer Science Press, 1976.

[9] Y. M. Levendel, P. R. Menon, and S. H. Patel, "Special purpose computer for logic simulation for distributed simulation," *Bell Systems Technical Journal*, vol. 61, pp. 2873–2909, Dec. 1982.

[10] R. Raghavan, J. P. Hayes, and W. R. Martin, "Logic simulation on vector processors," in *Proc. of International Conference on Computer Aided Design*, Nov. 1988.

[11] D. R. Jefferson, "Virtual time," *ACM Transactions of Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[12] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message Based Parallel Processing.* Potomac, Md.: Scientific Computation Series, MIT Press, 1987.

[13] J. McLeod, "Superfast simulators make it a lot easier to skip prototyping," *Electronics*, vol. 60, pp. 61–62, May 1988.

[14] K. M. Chandy and J. Mishra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions of Software Engineering*, vol. SE-5, pp. 440–452, Sept. 1979.

[15] J. Mishra, "Distributed discrete event simulation," *ACM Surveys*, vol. 18, pp. 39–65, Mar. 1986.

[16] G. Pfister, "The yorktown simulation engine," in *Proc. of 19th ACM/IEEE Design Automation Conference*, June 1982.

[17] D. K. Beece, G. Papp, and F. Villante, "The ibm verification engine," in *Proc. of 25th ACM/IEEE Design Automation Conference*, June 1988.

[18] Zycad-Corporation, "The zycad logic evaluator: Product descripition," *North Roseville, Minnesota*, 1983.

[19] P. Agrawal, "Mars: A multiprocessor based programmable accelerator," *IEEE Design and Test of Computers*, vol. 4, Oct. 1988.

[20] P. Agrawal, W. J. Dally, and R. Tutunjian, "Algorithms for accuracy enhancements in a hardware logic simulator," in *Proc. of 26th ACM/IEEE Design Automation Conference*, June 1989.

[21] N. Coleman and T. Ambler, "The event processor - a general purpose accelerator," in *CAD Accelerators* (P. A. Tony Ambler and W. Moore, eds.), New York: Elsevier Science Publishers B.V., 1991.

[22] C. E. Stroud, R. R. Munoz, and D. A. Pierce, "Cones: A system for automated synthesis of vlsi and programmable logic from behavioral models," in *Proceedings of IEEE International Conference on Computer Aided Design*, pp. 428–431, Nov. 1986.

[23] A. Saha and M. D. Wagh, "Algorithms for determining optimal partitions in parallel divide-and-conquer computations," in *Proc. Int'l Conf. Par. Proc.*, (St. Charles, IL), Aug. 1991.

[24] H. Sethu and M. Wagh, "Design of time optimal hardware-efficient divide and conquer algorithms," in *Proc. of Intl. Conf. on Parallel Processing*, (St. Charles, IL), Aug. 1992.

[25] R. Neogi and M. Wagh, "Processor assignment algorithms for parallel simulation of irregular task graphs," in *Proc. of 1993 SCS Simulation Multiconference*, (Arlington, VA), Apr. 1993.

[26] S. Bokhari, "On mapping problem," *IEEE Transactions of Computers*, vol. 30, Mar. 1981.

[27] V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," in *Proc. of 1986 Symposium on Compiler Construction*, (New York, NY), July 1986.

[28] B. Ackland and Kravitz, "Cemu - a concurrent timing simulator," in *Proc. on Intl. Conf. on Computer Aided Design*, (Santa Clara, CA), Nov. 1985.

[29] P. Agrawal, "Concurrency and communication in hardware accelerators," *IEEE Transactions of CAD of Integrated Circuits and Systems*, vol. 5, Oct. 1986.

[30] C. A. Petri, "Communication with automata," Tech. Rep. RADC-TR-65-377, Griffis Air Force Base, 1966.

[31] J. L. Peterson, "Petri nets," *ACM Comput. Surveys*, vol. 9, pp. 43–59, Sept. 1977.

[32] T. Murata, "Petri nets: Properties, analysis and applications," *IEEE Transactions of Software Engineering*, vol. 77, pp. 541–580, Apr. 1989.

[33] M. K. Molloy, *On the integration of delay and throughput measures in distributed processing models.* Ph.D. dissertation, UCLA, 1981.

[34] K. Jensen, "Colored petri nets and the invariant-method," *Theoretical Computer Science*, vol. 14, pp. 317–336, 1981.

[35] T. Smigelski, T. Murata, and M. Sowa, "A timed petri-net model and simulation of dataflow computer," in *Proc. of Intl. Workshop on Timed Petri nets*, (Torino, Italy), pp. 67–73, July 1985.

[36] M. A. Holiday and M. K. Vernon, "Performance estimates for multiprocessor memory and bus interference," *IEEE Trans. of Computers*, vol. 36, pp. 76–85, Jan. 1987.

[37] M. Diaz, "Modeling and analysis of communication and cooperation protocols using petri net based models," *Computer Networks*, vol. 6, pp. 419–441, Dec. 1982.

[38] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems," *ACM Trans. on Comp. Systems*, vol. 2, pp. 93–122, May 1984.

[39] M. A. Marsan, G. Balbo, G. Chiola, and G. Conte, "Generalized stochastic petri nets revisited: Random switches and priorities," *International Workshop on Timed Petri Nets*, pp. 49–59, Aug. 1987.

[40] M. A. Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems.* Series in Computer Systems, Cambridge, Mass.: MIT Press, 1986.

[41] S. Karlin and H. M. Taylor, *A First Course in Stochastic Processes.* New York: Academic Press, 1975.

[42] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* New Jersey: Prentice-Hall, 1982.

[43] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1986.

[44] M. Carlton and P. V. Roy, "A distributed prolog system with and parallelism," *IEEE Software*, vol. 5, pp. 43–51, Jan. 1988.

[45] J. T. Rayfield and H. F. Silverman, "System and application software for the armstrong multiprocessor," *IEEE Computer Magazine*, vol. 21, pp. 38–52, June 1988.

[46] V. D. Agrawal and S. T. Chakradhar, "Performance analysis of synchronized iterative algorithms on multiprocessor systems," *IEEE Transactions of Parallel and Distributed Systems*, vol. 3, pp. 739–746, Nov. 1992.

[47] J. D. Ullman, *Computational aspects of VLSI.* Rockville, Md.: Computer Science Press, 1984.

[48] A. K. Ahluwalia and M. Singhal, "Performance analysis of the communication architecture of the connection machine," *IEEE Trans. Parallel and Dist. Systems*, vol. 3, pp. 728–738, Nov. 1992.

[49] G. Ciardo and K. S. Trivedi, "Solution of large gspn models," in *Numerical Solution of Markov Chains* (W. J. Stewart, ed.), New York: Marcel Dekker, 1991.

[50] I. Mitrani, "Fixed-point approximations for distributed systems," in *Mathematical Computer Performance and Reliability* (G. Iazeolla, ed.), Amsterdam: North-Holland, 1984.

[51] H. Choi and K. S. Trivedi, "Approximate performance models of polling systems using stochastic petri-nets," in *Proc. of 11th. Ann. Jnt. Conf. on Com. and Comm.*, (Florence, Italy), pp. 214–221, May 1992.

[52] A. O. Allen, *Probability, Statistics and Queuing Theory with Computer Science Applications.* San Diego, CA: Computer Science and Scientific Computing, 1978.

[53] T. M. Niermann, W. T. Cheng, and J. H. Patel, "Proofs: A fast, memory efficient sequential fault simulator," *IEEE Transactions of CAD*, vol. 11, pp. 198–207, Feb. 1992.

Glossary of Symbols

Chapter 3

G_d	Input task graph.
V_d	Nodes of input task graph.
E_d	Edges of input task graph.
G_p	Processor graph.
V_p	Nodes of processor graph.
E_p	Edges of processor graph.
N	Total number of processors in the multiprocessor system.
K	Total number of nodes in the task graph.
λ	Unit computation cost relative to unit communication cost.
γ	Load-deviation bias constant.
$CCC(d,p)$	Cum. communication cost for assigning node d to processor p.

$LD(d,p)$	System load deviation cost for assigning node d to processor p.
$F(d,p)$	Total cost for assigning node d to processor p.

Chapter 4

$L1$	List of active signals.
$L2$	List of active blocking-devices.
α	Survival probability in 2-dimensional funneled pipeline.
μ_1	Average processing speed of stage-1 in 2-dimensional funneled pipeline.
μ_2	Average processing speed of stage-2 in 2-dimensional funneled pipeline.
$T_{2Dfunnel}$	Throughput of 2-dimensional funneled pipeline.
λ	Average speed of L2-processor in 1-dimensional funneled pipeline.
η	Average speed of L1-processor in 1-dimensional funneled pipeline.
m	number of L2-processors in 1-dimensional funneled pipeline.
T_{comp}	Throughput of 1-dimensional funneled pipeline.

Chapter 5

$1-u$	Probability of getting injected into the synchronization network.
ϕ	Average queue length of router processing element buffer.
Λ	Throughput in synchronization network.
P_{ref}	Probability of referral in synchronization network.
Ω_{tot}	Average total number of hops in synchronization network.
T_{locQ}	Expected waiting time in local queue of router processing element.
R	Average response time in the synchronization network.
ψ	Average processing speed of global processing element.
δ	Average message arrival rate in the global processing elements' waiting room.
T_{system}	System throughput.
H	Hardware specialization factor.
S	Speedup due to acceleration.

Vita

Raja Neogi, son of Mr. K. M. Neogi and Mrs. Krishna Neogi, was born in Calcutta, India on September 23, 1961. He secured his B.E in 1985 in Electrical Engineering from the Bengal Engineering College (Calcutta University) India and his Masters Degree in 1988 in Electrical Engineering from the University of Illinois, Chicago. He began his doctoral work upon joining the Department of Electrical Engineering and Computer Science at Lehigh University, Bethlehem, in the year 1989. His research interests include vlsi-CAD, vlsi-design, parallel-processing, computer architecture and object-oriented software engineering.