On Metrics for the Dynamic Load Balancing of Optimistic Simulations

Khalil El-Khatib and Carl Tropper
School of Computer Science
McGill University, Montreal, Canada, H3A 2A7
{kkhatib, carl}@cs.mcgill.ca

Abstract

The research described in this paper focuses on evaluating metrics for use with the dynamic load balancing of optimistic simulations. We present a load balancing algorithm in this paper which is token based and is used in conjunction with Clustered Time Warp (CTW). CTW is a hybrid synchronization protocol, which makes use of a sequential algorithm within clusters of LPs and Time Warp between the clusters.

We define three separate metrics and measure their effectiveness in different simulation environments. One metric measures processor utilization, a second measures the difference in virtual times between the clusters, while a third is a combination of these two metrics. We compare the execution time, memory consumption and the throughput obtained in three simulation environments by each of these metrics and to the results obtained without load balancing. Our categories of simulation are vlsi simulations, characterized by a large number of LPs and a low computational granularity; distributed network simulations, in which the workload varies spatially over the execution of the simulation; and a pipeline simulation, characterized by a single direction of message flow.

The experiments revealed a significant improvement in the simulation times in the first two categories of simulations when we employed the processor utilization and the combination metrics. For example, improvements of up to 70% were obtained for vlsi simulations. None of the metrics proved to be effective for the pipeline simulation.

1. Introduction

In order to minimize the execution time of a parallel simulation or indeed any parallel program, it is necessary to efficiently partition the modules of the program among processors. The mapping may be accomplished in either a static or dynamic fashion. In the static approach we assume a priori knowledge of the execution time and communication patterns of the program, and the module to processor mapping is decided ahead of run-time. The dynamic approach moves modules between processors periodically, basing its decision on conditions which arise during the execution of the program.

A number of dynamic load balancing algorithms have been developed for use with both conservative and optimistic algorithms, e.g. [3], [8], [9]. Section two of the paper contains a brief overview of work on dynamic load balancing algorithms.

The choice of a metric for use with a dynamic load-balancing algorithm is dependent upon a number of factors, among which are the nature of the model being simulated, the choice of the particular simulation algorithm and the platform upon which the model is simulated. Consequently, it is worthwhile to evaluate different metrics for dynamic load balancing in different simulation environments. The research which we describe in this paper is directed towards this goal.

We focus on the load balancing of optimistic simulations. Readers unfamiliar with optimistic simulation might wish to consult [14] for a background in this area.

We make use of a token-based dynamic load-balancing algorithm which is described in section three. We implemented the algorithm in conjunction with Clustered Time Warp (CTW) [3], [4], and [5], in which, as the name implies, the LPs involved in a simulation are grouped into clusters. A sequential algorithm is made use of within each cluster, and Time Warp is used between clusters- hence the clusters play a role analogous to that of the individual LPs in a conventional implementation of Time Warp. The original motivation for developing CTW was for use in parallel logic simulation, characterized by a large number of LPs and a low computational granularity. CTW made a particularly appealing environment for experimenting with distributed load balancing algorithms because the LPs are presumed to be organized into clusters at the beginning of the simulation and CTW performs cluster migration.

The algorithm makes use of a token, which circulates among the clusters, connected in the form of a virtual ring and collects information pertaining to the load on each cluster. It attempts to balance the load among the processors, which is defined via three separate metrics.
One measures the processor utilization; a second measures the discrepancy in virtual time between clusters, while a third metric is a combination of these two metrics.

The primary focus of our experimental work was to determine which of these metrics proved to be the most effective for different categories of simulations. We examined their utility in: (1) Logic level vlsi simulations, characterized by a large number of LPs and a low computational granularity. Other challenges posed by these simulations include the presence of a large number of feedback cycles and the fact that only a small percentage of the LPs involved in the simulation are active at any given time (2) Distributed network simulations. We investigated the effects of varying workload both in time and in space, i.e. different parts of the network were active at different points in time. We also investigated a uniformly distributed workload (3) Pipeline simulations, characterized by a single direction of message flow.

The remainder of this paper is organized along the following lines. Section two contains an overview of previous attempts at dynamic load balancing, section three contains a brief description of CTW, section four contains a description of the algorithm, section five contains a description of our experiments and results and finally, section six contains the conclusion.

2. Load Balancing

In the general partitioning problem, one is given a parallel (or distributed) computer system and a parallel (or distributed) program composed of modules which communicate with each other. It is required to map the modules onto the processors in such a way as to minimize the execution time of the program. The mapping can be categorized as being either static or dynamic. The static approach assumes a priori knowledge of the execution time and communication pattern of the simulation, and the task-to-processor mapping is decided ahead of run-time. The dynamic approach, in contrast, moves modules between processors whenever this leads to improved efficiency.

In this section, we describe previous attempts at dynamic load balancing. For a review of work on static partitioning, the reader might wish to consult [17].

The literature is rich in general-purpose dynamic load balancing algorithms [20], [13], and [2], but only a handful of these algorithms apply to PDES.

In [23], the author presents a simple dynamic load balancing for conservative simulation. Three phases take place in his dynamic allocation scheme. In the first phase, the process(es) which need allocation are identified-these are the processes whose service times fall outside a predefined interval of service time. In the second phase, process(es) which should be migrated are identified-processes identified in phase one which have predecessors or successors on different processors are considered for reallocation. The third phase is for identifying the processors to which the process(es) are reallocated: processors containing successors or predecessors for selected processes will be chosen first. The algorithm was tested on two logical systems (pipelines) on an iPSC Hypercube with 4 nodes. Experiments showed that when using the load balancing algorithm the running time of the simulation is reduced only when the system exhibits a non-uniform distribution of messages.

Boukerche and Das [6] presented a dynamic load balancing algorithm for conservative simulation using the Chandy-Misra [10] null message approach. Their algorithm uses the CPU queue length as a metric, as a way of representing the workload at each processor. In their approach, a Load Balancing Facility (LBF) is implemented in two separate phases: load balancing and process migration. In the first step of the algorithm, every processing element in the simulation sends the size of its CPU queue length to a central processor in which processors are classified according to the deviation of their respective queue lengths from the mean. The second step for the load balancer is to select the heavily overloaded processes from the heavily overloaded processors. Experiments were conducted on an Intel Paragon A4. The authors chose an N*N torus queuing network model with an FCFS service discipline as a benchmark in their experiments. Their results showed approximately a 25-30% reduction in the simulation time using dynamic load balancing over a random static partitioning when 2 processors were employed. A reduction of 40% was observed when the number of processors increases from 4 to 8. Similarly, the authors observed a 50% reduction in the run time when changing the number of processor from 8 to 16. The authors also reported a reduction in the synchronization overhead1 with dynamic load balancing: when less than 4 processors are used, the reduction was approximately 25-30% in the synchronization overhead. When 8 and 16 processors were used, the reduction was 10-40%.

Burdorf and Marti [8] deal with the problem of dynamic load balancing for the RAND Time Warp system. The system runs on a set of workstations shared with other users, which creates a large variation in the loads depending on the number of users and processes. Initially, a static balancer assigns objects to processors according to the load, which is gathered by running a pre-simulation. During the simulation, the balancer

---

1 Defined as the number of null messages processed by the simulation using the Chandy-Misra null-message approach divided by the number of real messages processed.
records the smallest simulation time of all objects on each machine. To minimize rollbacks, the dynamic load balancer seeks to minimize the variance between the objects’ simulation times. The authors presented four schemes for load balancing. Three of them use Local Virtual Time (LVT) as a metric, and the fourth uses the ratio of total processed messages to the total number of rollbacks. The algorithm was implemented on a simulation of colliding shapes moving in a constricted space. The authors report that the best results were achieved when objects which are furthest ahead are moved to machines which are furthest behind, and objects which are furthest behind are moved to machines which are furthest ahead. Results using the dynamic load balancing strategy showed a five to 10 times performance improvement over simulation with only static balancing.

Schlagenhaft et al [22] describe a dynamic load balancing algorithm for Time Warp VLSI circuit simulation. The algorithm consists of three components: (1) load sensor (2) load evaluator and (3) load adapter. The load sensor computes the Virtual Time Progress (VTP) for each simulation process.

The load sensor first calculates the Integrated Virtual Time (IVT) at the end of each simulation step. The IVT of a processor is defined as the average of all of the virtual times of the clusters residing on that processor. The VTP during a time interval \([T_1, T_2]\) is then computed as the change in the IVT per unit real time. The load evaluator decides whether to launch process migration or not, depending on ratio between the actual and the predicted VTP; if the ratio is big enough (migration is worthwhile), then load balancing is initiated. Process migration is then controlled by the load adapter. The authors simulated one logical circuit ([s13207 [7]]) on two processors shared with other users. Corolla partitioning was used to partition the circuit into small clusters with many of them mapped to each processor. The results showed that the algorithm reduces the lag between the VTPs of the processors which resulted in an increase in the advance of the GVT, and hence a 24% reduction in the simulation time.

Reiher and Jefferson [21] presented a dynamic load balancing algorithm for the Time Warp Operating System (TWOS). Their algorithm was tested on battlefield simulation in which objects are created and destroyed concurrently. For this implementation, dynamic load balancing was necessary in order to ensure good assignment of the new object to processor, and to balance the load after some objects have been destroyed. In their simulation, the life span of an object is divided into phases, which can run on different processors. Each phase is responsible for handling an object’s data and variables for one interval of virtual time. The algorithm tries to balance the load using an estimate of effective work, defined as the portion of the work that will not be rolled back. The authors tested their algorithm on two simulation models, a military simulation and two-dimensional colliding pucks simulation. Their results showed that speedups for the pucks simulation are equal to or less than the normal speedups because of the relatively even balance of pucks. Because of the inherent imbalance in the battlefield simulation, the dynamic load balancing algorithm improved the speedup by 25%.

Glazer and Tropper [11] introduced the notion of a time slice and simulation advance rate (SAR) in their work on dynamic load balancing. The purpose of the dynamic load balancer was to reduce the number of rollbacks, and hence reduce the simulation time. To this end, the SAR is computed at each processor and sent to one dedicated processor. The SAR is defined as the advance of the simulation clock divided by the CPU time needed for the advance. The load of a process, derived from its SAR, is a measure of the amount of CPU time it requires to advance its local simulation clock by one unit. The length of the time slice for a process is determined by its load: each process is given time slice proportional to the ratio of its load to the mean of all of the loads. To evaluate the performance of the algorithm, three different simulation models were constructed, representing different classes of models: a pipeline model, an hierarchical network model and a distributed network model. Experiments were conducted on PARALLEX, an emulation of a parallel simulation on a uniprocessor machine. The authors reported a 16-33% decrease in rollbacks and 19-37% increase in the simulation rate when only 8 processors were used. Results showed better improvement when the number of processors was increased: 42-71% and 47-49% decrease in the rollbacks when using 16 and 32 processors respectively, in addition to 30-39% and 49-65% increase in the simulation rate.

Recently, Carothers and Fujimoto [9] proposed a load distribution system for background execution of Time Warp. The system is designed to use the free cycles of a collection of heterogeneous machines to run a Time Warp simulation. Their load management policy consists of two components: the processor allocation policy and the load balancing policy. The processor allocation policy is used to determine the set of processors that can be used for a Time Warp simulation. This set is computed dynamically throughout the simulation. A processor is added or removed from the set when an estimate for the Time Warp execution time on that processor falls below or moves above certain thresholds. As in [11], the metric used was the Processor Advance Time (PAT), which reflects the amount of real time used to make a one unit advance in virtual time. Using the centrally collected PATs, the load balancing policy tries to distribute the load evenly over all processors. The algorithm was implemented on a set of
nine Silicon Graphics machines, one of which was dedicated to dynamic load management. The Time Warp application used in the experiments is a simulation of a personal communication services network. The authors employed a time-varying external workload throughout the course of the experiments. In one set of experiments, half of the processors experienced a monotonic increase in the external workload resulting in cluster migration to other processors. Results showed an improvement in the simulation time of up to 45%. To see how the system reacts to the change in the usable set of processors, a "spike" workload was added to four of the eight workstations. The authors observed a small improvement in the simulator efficiency and a reduction of the simulation time. This minor improvement was believed to be the result of the extra overhead of considering inactive processors in the computation of GVT. A delay in the GVT computation would result in a memory shortage on active processors since memory is not reclaimed fast enough.

Avril and Tropper [4] studied dynamic load balancing for Clustered Time Warp. To measure the load, the authors defined the "load of a cluster" to be the number of events which were processed by the cluster since the last load balance in the simulation. The load of a processor is then computed as the sum of all of the loads of all of the clusters residing on the processor. A processor called the pilot is made use of to collect load information from all of the processors; the load of each processor is piggybacked on the GVT token, and is forwarded to the pilot. The algorithm then iteratively chooses the most heavily and lightly loaded processors and transfers the cluster whose load is closest to half of the difference between the loads of these two processors. The authors describe a triggering technique based on the throughput\(^2\) of the system: the load balancing algorithm is triggered only when overall increase in the throughput of the system becomes larger than the cost (in terms of throughput) of moving the clusters. The algorithm was implemented and its performance was measured using two of the largest benchmark digital circuits of the ISCAS'89 series [7]. Results showed an improvement of 40-100% in the throughput of the system when dynamic load balancing was used. The authors observed that minimizing interprocessor's communications when moving clusters reduces the number of rollbacks, but does not improve the throughput.

3. Clustered Time Warp (CTW)

An implementation of our algorithm for load balancing was developed for Clustered Time Warp (CTW) [3], [5]. CTW is a hybrid algorithm which makes use of Time Warp between clusters of LPs, and a sequential algorithm within the cluster. The original motivation for CTW was logic simulation, characterized by its large memory demands and low computational granularity.

To reduce memory consumption and to avoid cascading rollbacks, CTW assembles many LPs into one cluster. Each cluster in Clustered Time Warp is autonomous: it receives messages, processes events, saves states and messages, and rolls back in case of a straggler or anti-message. A cluster consists of one or more LPs (gates), a Cluster Environment (CE), a Timezone table, and a Cluster Output Queue (COQ). The COQ holds the output messages of the cluster and is used in case of rollback. The CE is the manager of the Timezone table and the COQ. Figure 1 shows the complete structure of a cluster with four LPs.

Initially, the cluster timezone consists only of the interval \([0, +\infty]\), and is updated over time. When a cluster receives an event with a timestamp \(t\), it determines which timezone \(t\) fits into, and divides it into two timezones, \([t, t]\) and \([t, t]\).

Each LP in the cluster keeps its Local Simulation Time\(^3\) (LST), as well as the time of the last event it processed (TLE). When the LP processes an event, it determines if its timezone is different from the one of the TLE; if this is the case, the LP saves its state. We refer to this manner of recording checkpoints as Cluster Checkpointing (CC).

When an LP tries to send an event, it checks the receiving LP; if this LP resides on the same cluster, the message is inserted into the input queue of the receiving LP; if not, the message is handed to the CE which takes care of forwarding the message to the cluster in which the receiving LP is located.

\(^2\) Defined as the number of non-rolled-back message events per unit time.

\(^3\) Local Simulation Time is the same as local virtual time.
When a cluster receives a straggler with timestamp $t_s$, the CE creates a new timezone, and rolls back all of the LPs with TLE greater than $t_s$. After the rollback, a "coast forward" is executed at the LP level. The cluster behaves similarly when it receives an anti-message, except that it will not create a new timezone.

In this approach, LPs which are not directly affected by the straggler or the antimeessage are also rolled back. In order to avoid these unnecessary rollbacks, a new scheme called Local Rollbacks (LR) was devised. In this scheme, when a cluster receives a straggler or an antimeessage, it updates its timezone table, and forwards the event to the input queue of the receiving LP. In this way, only the receiving LP will be rolled back.

Another variant of CTW deals with checkpoint timing. Instead of saving its state when it enters a new timezone, an LP saves its state each time it receives a message from an LP located in a different cluster. Even though the new scheme decreases the number of states saved, there is an increase in the number of events an LP has to keep. These events are needed for coast forward, when an LP is rolled back to a state prior to the GVT. This new scheme is called Local Checkpointing, as opposed to Cluster Checkpointing.

Using the previous schemes, three checkpointing algorithms were developed: Clustered Rollback Cluster Checkpointing (CRCC), Local Rollback Local Checkpointing (LRLC), and Local Rollback Cluster Checkpointing (LRCC). These algorithms exhibit a trade-off between memory and execution time. CRCC uses the least memory and is correspondingly the slowest of the three algorithms, while LRCC uses the most memory and is the fastest of the three algorithms. LRLC exhibits a behavior which is intermediate between that of LRCC and CRCC. Experimental results about the performance of each of these algorithms may be found in [3], [5].

4. Metrics and Algorithm

In this section we present the details of our load balancing algorithm and its associated metrics along with a discussion of relevant design issues.

4.1. Metrics

Dynamic load balancing requires both a metric to determine the system load as well as a mechanism for controlling process migration. Ideally, the metric should not only be simple and fast to compute, but also effective. We discuss three such metrics - Processor Utilization (PU), and Processor Advance Simulation Rate (PASR) as well as a combination of the two, $PU/PASR$. We turn first to PASR.

If several (simulation) processes are interconnected, a discrepancy in their respective virtual times can result in an increase in the number of messages arriving in the past, and cause rollbacks. When a process is rolled back from time $t_1$ to time $t_2$, all work performed during this time period is discarded. System resources used during the corresponding real time interval could have been productively employed by other processes.

Controlling the rate at which processes advance their corresponding virtual times will minimize the difference between the virtual clocks, and as a consequence, reduce the number of rollbacks which occur in the simulation.

The virtual time of a processor is defined as the minimum virtual time of all the processes residing on that processor. A processor which has no events to process sets its virtual time to infinity.

For a system simulated in the real time interval ($t_{start}$, $t_{end}$), the Processor Advance Simulation Rate (PASR) defines the rate of advance in virtual time relative to real time. Let $t_1$ and $t_2$ be two real time values, with $t_2 > t_1$. Define $ST_t$ as the simulation time at real time $t$. Let $\Delta ST$ denote the change in the simulation time during the time interval $(t_1, t_2)$:

$$\Delta(ST)^{t_2}_{t_1} = ST_{t_2} - ST_{t_1}$$

The PASR is defined as:

$$PASR = \frac{\Delta(ST)^{t_2}_{t_1}}{t_2 - t_1}$$

A processor with a PASR higher than average is susceptible to being rolled back because it is ahead of other processors in virtual time. If it is slowed down, the frequency with which it is rolled back by other processors might well decrease. Figure 2 shows an example of two processors with different PASRs. A message $m$ sent from $P_1$ to $P_2$ will force $P_2$ to roll back to a virtual time previous to $vt_1$, since the timestamp of $m$ is $vt_1$. $P_2$ then has to cancel all the previous work done in the $(t_1, t_2)$ real time interval. Hence, moving some load from processors with high PASRs to others with low PASRs should speed up the slow processors and slow down the fast ones.

A number of researchers [12], [24], [25] feel that it is best to maximize the available parallelism in the system by keeping processor utilization as high as possible. For systems where no a priori estimates of load distribution are possible, only actual program execution can reveal how much work has been assigned to individual processors.
Let us define effective utilization [21] as the proportion of work done by a processor which is not rolled back. Unfortunately, it is impossible for a processor to determine the effective utilization at a given point in the simulation since it might rollback later and cancel all of the work that has been done. In [21] an estimate of the effective utilization is used for load computation. Consequently we make use of the processor utilization (PU), defined as the ratio of the processor’s computation time (in seconds) between $t_1$ and $t_2$ to $t_2 - t_1$:

$$ PU = \frac{\text{Computation time in } (t_1, t_2)}{t_2 - t_1} $$

Processor utilization allows for the fact that messages in the system might be of different size, and might require different service times. It also accounts for the fact that two processors might advance their virtual clocks by the same value, even if the computation time is different.

A combination of the two metrics, PU/PASR, was also tested in our experiments. The combination was intended to increase the utilization of the processors, while maintaining maximum advance simulation rate and minimizing the number rollback.

4.2. Algorithm

A difficulty in the load balancing of a distributed application is the absence of global information on the load of the system. We employ a (distributed) algorithm to collect the relevant information about processors’ loads in our load balancing algorithm.

Our algorithm uses a token which circulates to each processor on a logical ring. At each processor, the PASR, the PU and the PU/PASR (all referred to as LOAD later) are inserted into the token.

Assume that there are $n$ processors in the system. Initially, the token is launched by processor one ($P_1$). When it gets to processor $n$, data from all of the processors will be stored in the token. Processor $n$, called the host, will be able to identify overloaded and underloaded processors. The token will remain in processor $n$ for a period of time after which it is launched again. After the next round, when the token reaches processor $n-1$, data from all of the processors will be contained in it, and hence $n-1$ becomes then the next host.

At the end of each round, the host processor computes both the mean and the standard deviation of all of the LOADs. The new mean is then compared with the mean from the previous round (stored in the token). If the new mean is larger, this indicates that the performance of the system is improving (increase in the PU or the PASR), and the system is left intact. The host sets a count-down timer which triggers the token again after time $\beta$. If the new mean is smaller than the mean from the previous round, the host checks the standard deviation of the LOADs. If the standard deviation is found to be larger than a certain tolerance value, $\alpha$, then a new process to processor mapping is assumed to be necessary. The host matches the processor with largest load together with the least loaded one and sends a message to the over-loaded processor with the name of the destination processor to which it should transfer some of its load. The load to be moved is equal to half of the difference in the loads between the two matched processors. The standard deviation of the loads is computed again, and another pair of processors is matched. This process is repeated until the standard deviation is less than the value $\alpha$. When all migrations are completed, the host starts the timer for the next round of the token.

The pseudocode presented below contains the basic functions for the dynamic load balancing algorithm.

```latex
function Serve-Token (token) {
  Insert processor’s LOAD into the token
  if (token contains loads from all processors){
    Mean LOAD $= \sum_{i=1}^{n} LOAD_i$ / $n$ /* $n$ is the number of processors */
    if (Mean LOAD $\geq$ token.(previous Mean LOAD)) /* The performance of the system is improving */
      Set-Timer();
    else {
      $\text{StD} =$ (standard deviation of LOADs);
      if ($\text{StD} \leq \text{tolerance } \alpha$) /* $\alpha$ was taken to be .25, a value determined
        experimentally*/
        Set-Timer();
      else {
        matchProcessors();
        while (processors are still transferring processes) wait(); /*Processor can proceed */
      }
    }
  }
}
```

\footnote{We determine this time period experimentally.}
Designing a process migration system involves resolving a number of issues such as which cluster to move; when the dynamic load balancing algorithm should be invoked, and what the maximum number of processors which are allowed to transfer load (at the end of each round) should be. These issues are all interrelated, and their resolution depends on the simulated model. We approach these issues in the following paragraphs, turning first to the determination of the token period.

The algorithm periodically sends out a token to collect the necessary information from all of the processors, and at the end of each round the host decides whether a process migration is needed. The time interval $\beta$ between two consecutive token rounds (token period) should be long enough to allow the system to stabilize after the previous load balance, but should be short enough to prevent the system from being unbalanced for a lengthy period of time.

During the course of the experiments, a token period of 10 GVT computations was employed. Another parameter whose value is important to determine is $\alpha$. Transfer of clusters occurs only if the difference of the standard deviations is greater than $\alpha$. We used a value of $\alpha = .25$, determined experimentally. While both of these choices are model dependent, further experience with this algorithm should be able to clarify a range of acceptable values for both parameters.

There is a trade-off between achieving the goal of completely balancing the load and the communication costs associated with migrating processes since transferring clusters takes time. When determining which cluster to move, we choose the cluster with the highest LOAD so as to minimize the number of clusters moved, assuming that the load of the cluster does not exceed the intended load to move.

Theoretically, the load balancing algorithm should distribute the loads on all of the processors as uniformly as possible. However, migrations from too many processors can cause the system to become unstable. In our experiments, we observed that approximately 30% of the processors can launch migration at one time without jeopardizing the stability of the system. Consequently, we limited the number of migrations to this level.

5. Experimental Results

As indicated by the title, this section is devoted to a description of the models which we simulated and the results which we obtained using our load balancing algorithm and metrics.

The load balancing algorithm was implemented on top of Clustered Time Warp. In our experiments, we made exclusive use of the LRCC checkpointing technique which offers an intermediate choice in the memory Vs execution time trade-off. The simulations were executed on a BBN Butterfly GP1000, a shared memory multiprocessor machine.

Three classes of models, based on VLSI circuits, assembly pipeline and distributed networks were simulated. Each of these classes of models exhibits different characteristics which are useful in evaluating the metrics- (1) Logic-level vlsi simulations are characterized by a large number of LPs and a low computational granularity. Other challenges posed by these simulations include the presence of a large number of feedback cycles and the fact that only a small percentage of the LPs involved in the simulation are active at any given time (2) Distributed network simulations. We investigated the effects of varying workload both in time and in space, i.e. different parts of the network were active at different points in time. We also investigated a uniformly distributed workload (3)

---

5 The GVT computation was initiated every 3 seconds.
Pipeline simulations, characterized by a single direction of message flow.

### 5.1. VLSI logic level simulations

Two digital circuits from the ISCAS’89 [7] benchmark suite (Table 1) were selected and simulated. The size of each of these circuits is approximately 24,000 gates.

#### Table 1: Circuits from ISCAS’89

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Inputs</th>
<th>Outputs</th>
<th>flip-flop</th>
<th>Number of gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₂</td>
<td>38584</td>
<td>12</td>
<td>278</td>
<td>1,452</td>
</tr>
<tr>
<td>C₃</td>
<td>38417</td>
<td>28</td>
<td>106</td>
<td>1,636</td>
</tr>
</tbody>
</table>

The circuits were partitioned into 200 clusters each, using string partitioning [18]. Clusters were numbered arbitrarily and were mapped to processors as follows: if \( N \) is the number of processors, and \( K \) is the number of clusters \((K>N)\), then the first \( K/N \) clusters were assigned to processor number 1, the second \( K/N \) clusters were assigned to processor number 2, and so forth.

The data which was collected includes the running time, peak memory usage, and effective throughput. The performance of the algorithm was evaluated using between 12 and 24 processors on the Butterfly.

Figures 3 and 4 show the simulation time for the two circuits C₂ and C₃, plotted against the number of processors. Results are compared to the simulation without load balancing. The same number of input vectors was used for all of the simulations. Figure 3 depicts the performance of the metrics and the algorithm for C₂: 35-72% reduction in the simulation time when PU is used as a metric, 0-30% when PASR is used and 0-40% when \( PU^*(1/PASR) \) is used. For C₃, the results are as follows: 2-21% when PU is used as a metric, 5-16% when PASR is used and 0-15% when \( PU^*(1/PASR) \) is used. In circuit C₂ we see that large improvements occur when we employ 12-15 processors. In particular, the PU metric clearly produces far better results in this range of processors than does either PASR or \( PU/PASR \).

The performance benefits of the load balancing algorithm(s) decrease as we employ more than 16 processors. Similar behavior occurs for circuit C₃ although the improvement in performance caused by the load balancing algorithm is less marked.

We attribute the difference in the percentage of reduction in the simulation time between the two metrics to the locality of activity. When the system exhibits a high locality of activity, PU will increase on some processors. Underloaded processors can quickly process the events generated by a more heavily loaded processor resulting in the same PASR, but a different PU. This also explains why the improvements for C₂ decline with an increase in the number of processors since the activity of the circuit is spread out when using more processors.

To understand the difference in the operation of the load balancing algorithm between the two circuits, why dynamic load balancing may sometimes increase the running time of the simulation (employing PASR with C₃, we looked at two other elements: peak memory consumption and the effective throughput. The peak memory consumption is the average of the peak memory consumption in all of the processors. On each processor, it represents the maximum amount of memory used over the course of a run of the simulation. The effective throughput is defined as the number of non-rolled-back messages in the system per unit time.

Researchers on memory consumption for Time Warp [14], [19], [15], and [1] have pointed out that there is a time penalty associated with large space consumption, and that it is possible for a simulation to run out of memory. Memory management is time consuming; a heavily loaded processor must spend considerable time on memory management.
By looking at figures 5 and 6, one can see that dynamic load balancing reduced peak memory consumption for C₂, but increased it for C₃. Circuit C₂ has a higher locality⁷ of events than does C₃, hence moving clusters and associated events from the loaded processors in C₂ decreases the peak memory consumption. As for C₃, the activity of the circuit is much more distributed, and hence moving clusters might create a memory problem.

An increase in rollbacks was observed (up to 20%) when load balancing was employed. This increase was expected since moving clusters slowed down the source and the destination processors; when these processors resume computation they might well send messages into the “past” of processors which were not involved in moving clusters. In addition, it is sometimes the case that the virtual time of the migrated cluster is smaller than the virtual time of the receiving processor. This can result from the fact that the receiving processor is being lightly loaded, and is ahead in virtual time of the sending processor. The reason that the PU metric manages to obtain good results in the load balancing algorithm is that the increase in the number of rollbacks is more then compensated for by the balancing of processor utilizations.

The increase in the number of rollbacks is compensated for by the balancing of processor utilizations.

⁷ An experimental study about the activity of the same two circuits can be found in [5].
5.2. Pipeline Model

A second model which was simulated is a manufacturing pipeline (figure 9) [11]. The model consists of thirty processes, including two sinks and two sources, arranged in nine stages. Each process is a cluster of 625 logical processes connected in a mesh topology. Clusters at the same stage were mapped to the same processor. Messages in the system flow from sources to the sinks, following different paths. At each stage, the message is served and forwarded to the next stage, until it gets to the sink, where it leaves the system. The service time distribution is deterministic and the routing decision at each stage is governed by a uniform distribution. The pipeline model exhibits a large number of rollbacks which are caused by messages starting at the same source, following different paths, and arriving at the same processor in a (possibly) different order from the one in which they were generated.

Figures 10 and 11 show the results from the pipeline model. When dynamic load balancing was used, the simulation showed an increase in the percentage of rollbacks, and an increase in the simulation time. Figure 10 shows that the number of rollbacks increased by 20% when the PU metric was used, 18% when using the PASR and 17% when a combination of the two metrics was used. The increase in the percentage of rollbacks results from the fact that moving clusters from one stage (processor) to another will cause delay on some of the input links of the next stage (processor).

Figure 11 shows that the simulation time increased by 8% when using dynamic load balancing with the PU metric, 0.5% when using the PASR metric. When using the combination of the two metrics, the simulation time did not change. The increase in the simulation time is explained by the increase in the number of rollbacks in the system.

5.3. Distributed Network model

The final model is a distributed communication model (figure 12). Two kinds of experiments were
conducted on the model. In the first experiment, messages are uniformly distributed on the network. The second experiment modeled a national communication network divided into four regions. In this model, we experimented with the reaction of dynamic load balancing to a continuous change of loads on the processors. During the course of the simulation, messages were concentrated on different regions, one region at a time. For instance, at one point messages were concentrated in region 1, and regions 2, 3 and 4 were lightly loaded. After a period of time, region 2 became saturated with messages, and regions 1, 3 and 4 were lightly loaded.

The simulation runs on 10 processors, with 7-8 nodes mapped to each processor. Interprocessor communication was minimized by mapping the connected nodes to the same processor. On each node a message is served and with a probability of 30%, is forwarded to a uniformly selected neighbor. Nodes have service times governed by exponential distributions (with different means).

The results in figures 13 and 14 show a difference in the speedup percentage between the two experiments. Figure 13 shows a 30-35% reduction in the simulation time when dynamic load balancing was employed with all metrics. This comes back to the fact that, in the presence of time-varying load, the system is locally overloaded with messages, and cluster migration improves the performance of the simulation. A reduction of only 10-20% was observed in the absence of time-varying load (figure 14).

6. Conclusion

In this paper, we compared the performance of three metrics for use with the dynamic load balancing of optimistic simulations. The metrics were used with a token-based dynamic load balancing algorithm which was implemented in conjunction with Clustered Time Warp [3]. Clustered Time Warp, as the name implies, gathers LPs into clusters, and uses a sequential algorithm within clusters and Time Warp between clusters.

In order to measure the load on the processors we defined three metrics- processor utilization (PU), processor advance simulation rate (PASR), and a combination of these metrics. To evaluate the performance of the algorithm with each of the metrics, several models were simulated- logic level vlsi models, an assembly pipeline model and a distributed communication network model. Each of these models was chosen because of their different characteristics- the vlsi simulations because of a large number of LPs, low computational granularity and paucity of active LPs during the course of any simulation; the distributed network simulation because of the spatial variation of the workload during the course of the simulation; and the pipeline simulation because of the uni-directional nature of message flow. Experiments were carried out on the BBN Butterfly GP1000, a 32 node distributed memory multiprocessor. The simulation time, memory consumption and effective throughput were measured. The effective throughput is the number of non-rolled-back messages in the system per unit time. Results obtained using each of the metrics were compared to those obtained without load balancing.

In performing vlsi logic level simulations, two of the largest circuits from the ISCAS89 benchmark suite were simulated. Our results indicate that the PU metric yields better results then the PASR or the combination metric
for both the high activity (C_2) and low activity (C_3) circuits. An improvement of 35-72% was obtained for C_2 while 2-21% was obtained for C_3.

As for the pipeline model, the simulation time increased by 8% using the PU metric and decreased slightly (5%) with the PASR metric. These results were a consequence of a large increase in the number of rollbacks resulting with both metrics.

Two distributed network models were simulated. One depicted a uniform distribution of messages, while in the other the locus of message traffic moved during the course of the simulation. All of the metrics yielded good results for the uniform model (30-35% improvement) while only the PU metric produced good results (20% improvement) for the moving traffic model.

It is clear that the effectiveness of a dynamic load balancing algorithm depends strongly on the nature of the model which is being simulated. Important model characteristics include which affect a dynamic load balancing algorithm the event granularity, the amount and direction of the message traffic between clusters, the number of LPs, and their level of activity over the course of the simulation.

The fundamental direction of research suggested by the results described in this paper is the determination of appropriate metrics for a given combination of model characteristics and computing platforms. We hope to continue this line of research.

Reference


