THE UNIVERSITY OF CALGARY

Sequential Performance of PDES Algorithms

by

Roger J. Curry

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
September, 2005

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Sequential Performance of PDES Algorithms" submitted by Roger J. Curry in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Brian Unger,
Department of Computer Science.

Dr. Rob Simmonds,
Department of Computer Science.

Dr. Abraham Fapojuwo,
Department of Electrical Engineering.

Sept. 6/2005

Date

Abstract

In this thesis the sequential performance of parallel discrete event simulation (PDES) algorithms is explored. These algorithms were originally developed to speed up the execution of a single discrete event simulation by using multiple processors. Currently most sequential discrete event simulators are based on the Central Event List (CEL) algorithm. Excellent sequential performance of the Critical Channel Traversing (CCT) algorithm has been reported in the literature. This has motivated the investigation of channel based conservative PDES algorithms as an alternative approach to sequential discrete event simulation.

A synthetic workload model is used to compare the sequential performance of channel based conservative algorithms with several CEL-based algorithms. The results demonstrate that a channel based conservative algorithm can often achieve a two to three times speedup in event rate over the fastest CEL-based algorithm. The greatest performance improvements are observed in situations of low connectivity, high event density, and large lookahead. Under adverse conditions the performance of the conservative algorithm can be much worse than that of the CEL-based algorithm. The potential causes of good and poor performance of channel based conservative algorithms are examined in detail.

Acknowledgments

I would like to begin by thanking Brian Unger and Rob Simmonds for recruiting me into the TeleSim research group. It was there that I was introduced to the interesting and challenging field of parallel discrete event simulation. Being part of this research group was an experience that I will always be thankful for. Thank you to Rob Simmonds, for encouraging me to pursue graduate studies and for his continual support throughout the process.

I would like to acknowledge our postdoctoral student Cameron Kiddle for his help with my thesis. Cameron provided many insights into the sequential cost analysis for the CMB algorithm. Thank you to committee member Dr. Abraham Fapojuwo whose comments were very helpful in improving the final version of my thesis. Finally, I would like to thank my friends and family for their support and encouragement over the last three years.

Generous financial support was provided by the TeleSim Research Group which is funded by ASRA (Alberta Science Research Authority).

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ANR	-	Aggressive No Risk
CCT	-	Critical Channel Traversing
CEL	-	Central Event List
CMB	-	Chandy, Misra, Bryant
DES	-	Discrete Event Simulation
DVE	-	Distributed Virtual Environment
FF	-	Far Future
FIFO	-	First In, First Out
GVT	-	Global Virtual Time
HLA	-	High Level Architecture
LP	-	Logical Process
LVT	-	Local Virtual Time
NF	-	Near Future
PDES	-	Parallel Discrete Event Simulation
PHOLD	-	Parallel HOLD
VFF	-	Very Far Future

Chapter 1

Introduction

Simulation is a way of imitating, understanding, and predicting the behaviour of a real world system over time. There are applications of simulation in many problem domains. For example, simulation is used in the design and analysis of manufacturing systems, air traffic control scenarios, and telecommunication networks. Simulation can be used for performance evaluation, forecasting, and sensitivity analysis. In some situations a simple analytical model may provide the same information. However, real world systems are often too complicated to express or solve mathematically. Simulation is an appropriate tool for analyzing the behaviour of such complex systems. In addition, simulation can be used when experimenting with a real world system would be too costly or dangerous.

There are several approaches to simulation. *Discrete event simulation* (DES) is one method for modeling a system as it evolves over time. DES is primarily used to model systems where state changes occur at discrete points in time. It is possible to approximate state changes that occur continuously, but discretization may incur some error. *Continuous simulation* is an alternative to DES that uses iteratively solved differential equations to model state changes that occur continuously over time.

Most discrete event simulations are executed sequentially using the resources of a single computer. Parallel discrete event simulation (PDES) refers to the execution

of a single simulation using multiple processors. This allows completion of a single simulation run in less time. PDES is appropriate when minimizing the execution time for a single simulation run is critical, or when the model is too large to simulate using the resources of a single computer. For example, PDES techniques are used in emulation [33] were real-time execution is necessary.

In practice, simulation studies often involve executing many independent simulation runs. For example, multiple simulation runs are necessary for variance reduction, parameter studies, constructing confidence intervals, or when comparing different solutions to a problem. Parallel simulation techniques reduce the execution time of a single simulation run, but sequential simulation achieves better overall efficiency.

This thesis is focused on improving the speed of sequential DES simulation through the use of techniques originally developed for PDES.

1.1 Motivation and Objectives

In a DES, *events* are used to model changes in system state. Each event has an associated *timestamp* that indicates when the state change will occur. Most sequential DES programs employ a *central event list* (CEL) to order execution of events in the system. The CEL is implemented using a priority queue of events sorted by event timestamp. The event with the smallest timestamp is removed from the CEL and then executed. The execution of an event may result in the generation of new events which are then inserted into the CEL. Much of the research in sequential DES has focused on the implementation of the priority queue used to represent the CEL.

CCTKit is a parallel simulation kernel that implements the Critical Channel Traversing (CCT) algorithm [42]. The CCT algorithm is an extension of the channel based conservative PDES algorithms first proposed by Chandy and Misra [4], and Bryant [3]. Several simulation systems employing the CCT algorithm have shown very good sequential performance. The ATM-TN network simulator achieved a three

times greater event rate when using CCT than a CEL-based simulator implemented using a splay tree [41]. The IP-TN network simulator has performed up to four times better than a CEL-based simulator employing a heap [21]. This contrasts with most other PDES systems that perform poorly in sequential runs [38]. Little work has been done to explain why a system using the CCT algorithm can perform so well sequentially.

This thesis examines channel based conservative PDES algorithms such as CCT, to understand how they achieve better sequential performance than CEL-based simulators, and under what conditions this occurs.

A synthetic workload model will be presented and then used to compare the sequential performance of the algorithms. Several well-known CEL-based algorithms are compared with the channel based conservative PDES algorithms. The asymptotic behaviour of the PDES algorithms is analyzed and compared with that of the CEL algorithms. Six performance metrics are defined that facilitate empirical comparison of the algorithms in terms of cache behaviour, computational complexity, and event rate.

Improving the performance of sequential DES programs has numerous benefits. Faster sequential DES will allow scientists and engineers to test and compare design options in less time. Where it used to take weeks to compare alternative designs, simulation will someday enable these options to be evaluated on the fly. Improving the performance of sequential DES algorithms could also lead to improvements in PDES simulators since sequential DES algorithms are often part of larger parallel simulations. For example, a central event list approach is used to execute events which are scheduled in a *cluster task* in TasKit [42]. Fast sequential DES kernels would also benefit the distributed simulation community, since many distributed simulations are federates of smaller sequential simulations. For example, the high level architecture (HLA) is often used to combine independent sequential simulators [14]. By demonstrating the benefit of applying parallel simulation techniques to sequential

simulation, or simply using parallel simulation algorithms sequentially, it may suggest other opportunities to exploit algorithms originally developed for parallel execution to more general computer science problems.

1.2 Overview of Thesis

The remainder of the thesis is organized as follows. In Chapter 2 background material regarding discrete event simulation is presented. Relevant definitions and terminology are introduced, followed by a detailed description of the central event list algorithm. The chapter describes different priority queue algorithms that have been used to implement the central event list. Chapter 3 discusses issues related to the execution of a discrete event simulation in parallel. Further terminology is introduced and the logical process modeling methodology is explained. The chapter includes descriptions of various PDES algorithms, including the CMB and CCT channel based conservative PDES algorithms.

Chapter 4 provides a survey of synthetic workload models that have been used in the comparison of both sequential and parallel DES algorithms. A detailed description of the workload model used in this thesis is then presented. The next part of this chapter covers the experimental methodology, including a description of the metrics used in the comparison of the different simulation algorithms. The last part of this chapter presents a theoretical cost analysis for the performance of channel based conservative PDES algorithms.

Chapter 5 presents the experiments and corresponding results. The experiments are grouped according to the parameters of the synthetic workload model that were manipulated. Algorithms are compared using the performance metrics defined in Section 4.2.4. Chapter 6 provides a summary of the thesis and outlines the contributions made by this thesis. The summary also includes a discussion of potential future work.

Chapter 2

Sequential DES Algorithms

The central event list algorithm is used by many sequential simulators. Much of the research in sequential discrete event simulation has focused on the implementation of the priority queue data structure used in CEL algorithms. A wide range of priority queue implementations are possible; some implementations are general purpose and others have been developed specifically for implementing the CEL. Several of the CEL implementations in this chapter will be used as a baseline for comparison of the channel based conservative algorithms described in the Chapter 3.

This chapter introduces the terminology and concepts central to DES. The CEL algorithm is described in detail and then possible implementations of the priority queue are discussed.

2.1 Discrete Event Simulation

Discrete event simulation (DES) is a useful tool for modeling the behaviour of systems where state changes occur at discrete points in time. Before a system can be simulated using DES, it must first be mapped to a logical model. In sequential DES there are relatively few restrictions on how that is accomplished. A discrete event simulation requires three things; a representation of time, a representation of state, and some

way of changing or affecting the state as time advances. A simulation program could use a floating point variable to represent time, in addition to other variables for the system state. In DES, system state is affected by executing events. At a minimum an object that represents an event will contain information that indicates when the event will occur, called the event *timestamp*, and the *type* of event that will occur. The collection of all unprocessed events in the system is known as the *pending event set*. As events are executed, the representation of time must be updated to reflect the passage of time.

The world view of a discrete event simulation is usually described as *event-oriented* or *process-oriented*. In the event-oriented view described above, simulation behaviour is described in terms of events that are used to affect the state as time advances. In the process-oriented view behaviour is described in terms of interacting processes that wait for a specific period of time or block until some condition is true. Although not as efficient as a strictly event-oriented view, the process-oriented view can offer a more intuitive approach to modeling certain physical systems. A process-oriented simulator is normally implemented on top of an event-oriented simulation system.

Simulation can be confusing because there are often several systems of time under consideration. This is further complicated in parallel simulation, where different parts of the simulation may have advanced further than others. Fujimoto [14] gives the following definitions to distinguish the different systems of time.

- *Physical time* refers to the time in the physical system.
- *Simulation time* is an abstraction used by the simulation program to model physical time.
- *Wall-clock time* refers to time during the execution of a simulation program.

The following example illustrates the difference between these systems of time. A simulation program begins running at 1:30pm and finishes running at 1:45pm, the

elapsed wall-clock time is 15min. During that period of time, the simulation time advanced from 0s to 3600s, 1 hour of physical time was simulated.

In general it is desirable for simulations to execute as fast as possible. There may be some situations such as network emulation [33] where simulation execution is paced to correspond with the wall-clock or physical time.

Causality refers to the principle that one action or event leads to another. If a rock is thrown at a window, then the window is broken. The action of throwing the rock results in the window breaking. The window will not break unless some force causes it. In a simulation, certain events may depend on previous events, and a simulator must ensure causal relationships are maintained. Simulators are constrained such that processing of an event with timestamp t must only result in scheduling new events with timestamps greater than or equal to t. Violation of causality can result in errors in a simulation. Such errors are called *causality errors.*

Simulations can be classified according to the mechanism by which simulation time is advanced. There are two time flow mechanisms, *event-driven* and *time-stepped.* The choice of time-flow mechanism also affects how causal relationships are maintained within the simulator.

2.1.1 Time-Stepped Time Flow Mechanism

Simulations that use a time-stepped time flow mechanism require that simulation time is split into small equal sized intervals. The simulation clock is advanced by a fixed increment and then all state variables are updated. If events are used to indicate state changes then all events with timestamps less than the simulation time are executed. Events which are executed during the same time slice are considered to be independent and to occur simultaneously, this may or may not be an issue. In order to preserve causal correctness the time-slice must be small enough to prevent dependent events from occurring out of order.

In time-stepped simulation the simulation time generally advances by the same

amount, this contrasts with event-driven simulation where simulation time advances in arbitrary increments according to the timestamp of each event. The size of the time-step is important because it will determine the simulation's precision with respect to time. Time-stepped synchronization has some applications in logical circuit simulation and has also been incorporated into numerous parallel DES algorithms. More information about time-stepped simulation can be found in [14]. This thesis focuses on algorithms which employ event-driven time flow mechanisms, described in the next section.

2.1.2 Event-Driven Time Flow Mechanism

The central event list (CEL) algorithm is an example of an algorithm that employs an event-driven time flow mechanism. A simple way to maintain causal correctness is by processing events in chronological order. The event with the smallest timestamp is removed from the pending event set and then executed. Executing this event may result in the generation of new events which are then added to the pending event set. This approach to synchronization is known as the central event list algorithm. A CEL-based simulator maintains causal relationships by using a single priority queue to order the execution of all events.

2.2 CEL

The majority of sequential simulation programs employ the CEL algorithm described in the previous section. Variations of the CEL algorithm are confined to the implementation of the priority queue. A priority queue is an abstract data type that supports at least two operations, *remove_min* and *insert*. The *remove_min* operation removes the element from the collection with the highest priority, while the *insert* operation adds an element to the collection. In this section the CEL algorithm is explained in detail and possible implementations of the priority queue are discussed.

In a CEL-based simulation the central event list corresponds to an instance of the priority queue, i.e., a class supporting operations `remove_min` and `insert`. The CEL algorithm operates as follows. The event with the smallest timestamp is removed from the central event list. Next, the simulation time is updated to the timestamp of the event that has just been removed. Finally, processing for the event occurs which may result in the generation of new events and their insertion into the central event list. If a simulation end time is specified, then events are only inserted into the central event list if their timestamps are smaller than the specified end time. This process repeats until there are no events left in the central event list.

Figure 2.1 provides pseudo-code for the CEL algorithm. The current simulation time is stored in the `sim_time` variable, `event` is a reference to an object that represents an event and `cel` is a reference to a priority queue representing the central event list. The `process` procedure handles processing of the event and freeing of memory allocated for that event.

Central Event List (CEL)

```
while event= cel->remove_min();
  sim_time= event->time_stamp;
  process( event );
```

Figure 2.1: Central Event List Algorithm

Every CEL algorithm processes events in non-decreasing timestamp order which guarantees causal correctness of the simulation. The only possible variation is in the processing of events with the identical timestamp. There has been extensive research into the implementation of general purpose priority queues and those specialized for DES. Empirical comparisons of priority queue implementations are available in [27, 20, 2, 31, 26]. Analytical results relevant to priority queue implementations are available in [27]. The following subsections describe various priority queue implementations

that have been used in DES systems.

2.2.1 Linked List

A linked list of events sorted by timestamp was used in many early simulation languages such as GPSS [16] and SIMSCRIPT [39]. It is rarely used in modern simulation because it does not scale well with respect to the size of the pending event set. Although the remove_min operation is $O(1)$, the average and worst case insert times are $O(P)$ where P is the number of elements in the collection. The worst case scenario is that the entire list is traversed before the event can be inserted which requires P comparisons. The average case assuming events in the pending event set are uniformly distributed would require about half as many comparisons.

Consider a simple model where the number of events remains constant. Let E_i be an event and E_{i+1} the event generated by the execution of E_i. The timestamp increment distribution or *scheduling distribution* describes the difference in timestamps between E_i and E_{i+1}. The average case search can be improved by employing heuristics to determine whether to begin the search from the front or the back of the list. As explained by McCormack and Sargent [27], the average portion of the list that must be traversed to insert a new event can be determined by computing the distribution of the pending event set. This information can be used to decide which end of the list to start the insertion from. For example, if the timestamp increment distribution is uniform, normal, or Erlang then insertion should start from the back of the list. For timestamp increment distributions like the hyper-exponential, mixtures of exponentials, and certain Gamma distributions insertion should begin from the front. McCormack and Sargent [27] also showed that when the pending event set is a mixture of events scheduled by more than one distribution the insertion should start from the front of the list. There are other heuristics based on maintaining a pointer to the median element or computing the average of the first and last elements, and then using this value to determine were to start the search. If the timestamp of the

element to be inserted is smaller than the average, start from the beginning while if larger, start from the end. This approach known as the median pointer method [39, 8] eliminates 50% of the comparisons.

Another way to exploit knowledge of the timestamp increment distribution is to use multiple lists as is done in the process-oriented SIMSCRIPT II.5 [27]. A separate event list is maintained for each simulated process. Events that are repeatedly scheduled from the same distribution should be inserted according to their timestamp increment distribution. Events associated with simulated processes are usually scheduled by a mix of distributions, so insertion should start from the front of the list as explained by McCormack and Sargent [27].

The linked list is easily implemented and has performed well in situations where the event list is relatively small. This is due to the higher overhead associated with more complex algorithms. The linked list may have some application as part of a larger simulation system.

2.2.2 Indexed-List

Indexed list algorithms first proposed by Vaucher and Duval [39] are based on keeping an array of pointers into the list spaced according to some increment of time. In this case the linked list is logically subdivided into lists whose elements fall within a certain quantum. For example, the first sublist contains events timestamped 0 to 5s, the next list 5 to 10s, and so on. These algorithms require an overflow list for elements that cannot be placed in one of the finite number of sublists. There are serious problems if the time increment is too large or too small. If it is too large the number of elements in individual sublists grows too much while if it is too small, the overflow list will contain too many events. Increasing the number of sublists can help but this delays the pointer lookup and uses additional memory.

Franta and Maly [10] use a second dynamically managed set of pointers allowing somewhat better management of the length of individual sublists. Events are inserted

by computing the offset into the first table of pointers and using that result to find the pointer to the appropriate sublist which the event is then placed in. This algorithm still suffers the overflow problem associated with Vaucher and Duval's algorithm. The worst case running time of these algorithms is still $O(P)$, since a large portion of the elements can end up in the overflow list. In practice, they perform better than this but are very sensitive to the timestamp increment distribution.

2.2.3 Binary Search Indexed List (Henriksen's algorithm)

Henriksen's algorithm [17, 18] sizes sublists according to the number of events they contain, rather than the amount of simulation time the events span. This can be accomplished using a single array of pointers. Binary search is used to find the pointer to the sublist for an insert. Pointers are updated with the insertion of each event to constantly moderate the length and number of sublists. This algorithm avoids a large overflow list since all sublists are kept approximately the same length.

Henriksen's algorithm was used in GPSS/H [17]. This algorithm displays greater sensitivity to the timestamp increment distribution than the heap algorithm as evidenced in [27, 20]. The worst case for a single operation is $O(P)$, but time per operation amortized over sufficient operations is bounded by $O(\sqrt{P})$.

2.2.4 Heap

A heap is a complete binary tree meaning that each level of the tree is completely filled except possibly the last where any nodes in the last level of the tree must occur as far to the left as possible. The heap condition for a node requires that its key be greater than or equal to the key of each of its children (if it has any). A complete binary tree is a heap if and only if each of its nodes obeys the heap condition.

Insertion begins by placing the new node into the last level of tree as far to the left as possible. If the last level is full, then the node is placed in the next level deeper. If the heap condition is satisfied for the parent of the inserted node then the insertion

is finished. Otherwise, the inserted node is swapped with its current parent until the heap condition is restored. If the inserted node is swapped all the way to the top of the tree this will require $O(\log_2 P)$ operations.

Removing the minimum timestamped event is simple since it is just the root node of the tree. The last element in the heap is moved to the root of the tree to replace the element being removed. Starting with the root node the heap condition is tested, if it is satisfied then the remove_min operation terminates, otherwise the current node is swapped with its smallest child. This process continues until the heap condition is restored. If one swap is performed at each level of the tree this will require $O(\log_2 P)$ operations.

The heap algorithm can be optimized so that the heap is restored only once per pair of insert and remove operations. This is more efficient when the remove_min operation is followed by an insert operation.

Insertion of new events or removal of the minimum timestamped event requires in the worst and average cases $O(\log_2 P)$ comparisons. One concern when using a heap is that two events with the same timestamp will not necessarily be executed in first-in-first-out (FIFO) order. The heap can be modified to ensure the FIFO property if necessary, but this will incur some additional cost.

Porter and Simon [29] and Gonnet [15] show that when insertions are random the average insertion time is bounded by a constant. Empirical results suggest that heaps are relatively insensitive to the timestamp increment distribution [20, 31] which provides good motivation for their use in general purpose sequential simulation kernels.

2.2.5 Splay Tree

The splay tree [35] algorithm is based on the binary search tree. A binary search tree can be an efficient data structure with an average of $O(\log_2 P)$ operations required to access tree elements. However, a binary search tree can become unbalanced if the data being inserted is not uniformly distributed, in which case performance can degrade

to $O(\sqrt{P})$. There are numerous approaches to maintaining balance or correcting imbalance in a binary search tree.

The splay tree was designed with amortized efficiency in mind. The tree is permitted to be in an arbitrary state, but each operation seeks to improve the structure of the tree to facilitate faster future operations. The worst case for a single insert or remove_min operation is in $O(P)$, but the cost per operation amortized over any sequence of operations starting from an empty queue is bounded by $O(\log_2 P)$.

Unbalanced trees can be balanced by performing *rotations* which promote one of the child subtrees while demoting the previous root node and any descendant nodes. Splay trees avoid part of the balancing cost by avoiding the need to examine both halves of the tree. However, it can increase the number of rotations performed. One pointer rotation and one comparison are required per item visited during a search of the tree.

The main advantage of a self adjusting tree is its ability to handle skewed data, or usage patterns. The drawback of a self adjusting algorithm is that the cost of a single operation in a sequence of operations may be expensive.

Splaying involves a rotation applied to each node in the search path which effectively halves the length of the search to any node in that path. Splay trees have performed better than heaps even though their asymptotic bounds are the same [20]. One explanation for this is that the splay tree is reordered bringing smaller elements to the top each time, so that on average remove_min requires less than $O(\log_2 P)$ operations.

2.2.6 Calendar Queue

Calendar queues [2] are based on the way humans use desk calendars. In practice they can obtain $O(1)$ average performance. The calendar queue algorithm provides a unique solution to the overflow list problem associated with certain other algorithms such as Vaucher and Duval's Indexed-list [39]. A calendar queue stores events in

sorted linked lists, one for each "day" of the "year". An array is maintained which stores a pointer to each of the lists. The remove_min operation returns the first element in the current "day", or if the list is empty then the next day is checked and so on. An event is inserted by computing which "day" it will occur on, and inserting it into that day's list. The calendar queue handles events that are further than a "year" into the future by requiring that the remove_min operation check that the event belongs to the current year before it is removed. There are two variables which determine the structure of the calendar queue, the number of days and the length of a day. These variables must be adjusted to obtain good performance as the queue size grows and shrinks.

Performance of the calendar queue will be optimal when the number of events is equal to the number of days in the year and events are uniformly distributed across the days in the year. In this scenario, insert and remove_min operations can be performed in constant time. The heuristic employed by the calendar queue algorithm is to double the number of days when the number of events exceeds twice the number of days, or halve the number of days when the number of events drops below half of the number of days. Whenever a halving or doubling occurs, it restores the ideal ratio of number of days to number of events. Changing the number of days requires recomputing the length of a day, and reorganizing the array (copying). This operation is relatively expensive but should occur infrequently. Unfortunately, the heuristic does not guarantee that an individual "day" will not have an excessive number of events, or that many days may contain no events at all. The worst case scenario for removing the minimum element is $O(DaysInYear)$, and because $DaysInYear$ is at most $2 \times P$, the worst case is essentially $O(P)$. The worst case for insertion is also $O(P)$, assuming all events are scheduled for a single day in the year. $O(1)$ is an amortized bound on number of operations required for a resize operation. This means that the cost of a resize operation is constant in the number of events when averaged over the execution of all insert and remove_min operations.

The calendar queue algorithm does not handle the case when there are too many sublists and the queue is a constant size [32]. The only time a resize operation occurs is when the queue size is changing. As such, if the distribution changes but the queue size doesn't the calendar queue can perform quite poorly, though it might be possible to monitor the distribution and adjust bucket size dynamically.

2.2.7 Lazy Queue

The lazy queue [32] is another multi-list algorithm for a priority queue. The lazy queue is modeled after a human approach to scheduling, partitioning events into one of three intervals, the near future (NF), the far future (FF), and the very far future (VFF). Each interval of events is handled differently, the near future events are kept completely sorted, the FF events are partially sorted using a sorted array of unsorted months, and the VFF which is not ordered.

When a remove_min operation occurs the smallest event is removed from the NF, if the NF is empty the first month of the FF is sorted and moved into the NF. The cost of inserting an event depends on which interval it will be inserted into. The cost of inserting events into the NF depends on the implementation of the NF priority queue. Insertion into the FF is very efficient since a calculation yields the appropriate month, and then the event is inserted into an unordered sublist, requiring $O(1)$ operations. Insertion into the VFF is also done in constant time since it is maintained unsorted. As time advances so must the boundaries between NF and FF, and, FF and VFF.

The performance of the lazy queue algorithm depends greatly on the size of the NF, FF, and VFF. There are several algorithm parameters that bound the size of the NF, FF, and VFF. Additional parameters control how and when to restructure the queue. Resizing the queue is quite expensive but its cost is amortized over the relatively inexpensive insert and remove_min operations.

The unordered sublists of the FF contrast with the ordered sublists of the calendar queue, also the lazy queue reintroduces the overflow list (VFF) and associated

problems.

2.2.8 Other Priority Queue Implementations

There are many other algorithms that have been used to implement the pending event set. A few more are briefly discussed below.

The priority-tree or ptree is a type of binary tree that can be efficient when timestamps of newly inserted events are generally larger than those already in the tree. As explained by McCormack and Sargent's theoretical analysis [27] this is not usually the case which makes the ptree a poor choice for general purpose CEL algorithm.

Mauricio Marin [26] introduces tournament-based binary trees and demonstrates their utility as a priority queue implementation well suited to DES. Complete binary trees were a predecessor to binary heaps; both structures have the benefit of being relatively insensitive to the scheduling distribution.

Binomial queues have been tested as a priority queue implementation for DES, but according to Jones [20] binomial queues' behaviour is erratic, requires complex code and is rarely faster than splay tree implementations.

Skew heaps can perform very well and they were one of the first tree mechanisms that did not restrict the cost of a single operation to $O(\log_2 P)$ instructions. Instead the algorithm guarantees that the amortized cost for any sequence of operations that starts from an empty queue is $O(\log_2 P)$ [20].

2.3 Summary

This chapter has presented relevant background for DES. The central event list algorithm has been explained and possible implementations of the priority queue have also been discussed. Table 2.1 summarizes the asymptotic behaviour of the different algorithms. Henriksen's, heap, splay tree, and calendar queue implementations have

Algorithm	Worst	Average	Comments
linked list	P	P	The insertion cost for a linked list is on average $O(P)$, but it may be significantly more efficient to start the insertion from the front or back of the list, depending on the timestamp increment distribution [27]. The cost of a remove_min operation is O(1).
heap	$\log_2 P$	$\log_2 P$	The insertion cost for a heap is on average O(1) under the assumption that insertions are random [15, 29].
impr. heap	$\log_2 P$	$\log_2 P$	The improved heap avoids unnecessarily restoring the heap when a remove operation is immediately followed by an insert operation[27].
henriksen	P	$\sqrt{P}$	The cost of a single operation can be $O(P)$, but the amortized cost per operation is bounded by $O(\sqrt{P})$ [22]. Empirical evidence suggests much better performance in practice [20].
splay tree	P	$\log_2 P$	The cost of a single operation can be $O(P)$, but the amortized cost per operation is bounded by $O(\log_2 P)$ [35].
calendar queue	P	1	Informal analysis and empirical evidence suggest O(1) average performance [2].
lazy queue	$\log_2 P$	1	Informal analysis and empirical evidence suggest O(1) average performance. Worst case performance depends on the implementation of the near future and VFF [32].

Table 2.1: Asymptotic Bounds for priority queue operations

been incorporated into the test system that is described in Section 4.2.1. These CEL-based algorithms will be used for comparison with the channel based conservative PDES algorithms described in the following chapter.

Chapter 3

PDES Algorithms

Parallel discrete event simulation (PDES) systems use multiple processors to execute a single simulation run. This can greatly reduce the time required to execute a simulation. Several parallel simulations programs have achieved exceptional sequential performance, in particular simulators employing the critical channel traversing algorithm [41, 21]. This algorithm is an example of a channel based conservative synchronization algorithm.

This chapter introduces terminology and concepts relevant to PDES. Section 3.2 presents several conservative PDES algorithms including the Chandy-Misra-Bryant (CMB) and critical channel traversing (CCT) algorithms. For completeness, Section 3.3 covers optimistic and aggressive no risk algorithms.

3.1 Parallel Discrete Event Simulation

Parallel discrete event simulation (PDES) refers to the execution of a single simulation run in a parallel or distributed computing environment.

There are two major classes of application for PDES technologies, *analytical* simulations and *distributed virtual environments* (DVEs) [14]. Analytical simulations are used as problem solving tools to determine particular characteristics of a system.

Distributed virtual environments [14] are used for training and entertainment purposes. They use simulation technology to create computer generated worlds in which geographically separated users interact. Emulations also allow users to interact in a simulated environment, but their purpose is usually to conduct analytical experiments in a repeatable test environment [33].

Analytical simulations are run as-fast-as-possible, whereas DVEs and emulations need only keep up to wall-clock time. Simulations that are paced with wall-clock time must execute quickly to meet real-time deadlines. DVEs that incorporate interaction with human beings are called human-in-the-loop; ones that support interaction with real-world devices are called hardware-in-the-loop. There may be certain situations such as the case of human-in-the-loop DVEs, where missing the occasional real-time deadline is acceptable provided the user doesn't perceive the virtual environment to be unrealistic.

Fujimoto [14] identifies four benefits of parallel simulation: reduced execution time, geographic distribution, integration of simulators that execute on machines from different manufacturers, and fault tolerance. Parallel simulation can reduce the wall-clock time required for a single execution of a simulation. This is necessary for applications such as emulation where meeting real-time deadlines is critical. Parallel simulation also enables simulation of larger models than would be possible with the resources of a single system.

3.1.1 LP Modeling Methodology

Similar to sequential DES, PDES requires that the physical system be mapped to a logical model before it can be simulated. PDES techniques generally impose more restrictions on how this mapping is accomplished. Most approaches use an object oriented or encapsulated approach. The mapping process begins by identifying the different parts of the physical system based on their interaction. Different parts of the physical system affect one another's state through some type of interaction,

whereas a single part may affect its own state directly. These physical components can be mapped to logical components within the simulation. A simulation represents parts of the physical system using objects, or classes, or whatever modeling tools are available to the simulation programmer. In PDES literature these simulation objects are called *logical processes* (LPs). LPs do not modify one another's state directly. Instead, they cause state changes in another LP by scheduling events which will later affect that part of the system. Representing the physical system in this manner makes it easy to understand and validate the model.

Many PDES algorithms use *channels* to represent the logical connections between LPs that communicate. Channel behaviour is first-in-first-out meaning that messages are received in the same order they were sent. A channel from LP A to LP B allows LP A to schedule events for LP B. LP A is said to have an *output channel* connected to LP B, while LP B has an *input channel to LP A*. Channels are often present in pairs to allow bidirectional interaction, i.e., a channel from LP A to LP B and a channel from LP B to LP A.

3.1.2 Causality and Synchronization

Events that depend on previous events must be executed in a causally correct manner. An error that occurs as a result of events be executed out of order is referred to as a *causality error*. The synchronization problem (i.e., the problem of maintaining causal correctness) is significantly complicated when attempting to run a simulation across multiple processors.

Attempting to use the CEL algorithm described in Chapter 2 for parallel simulation is problematic. The central event list is a shared data structure that will be accessed by all processors. It must be locked whenever it is accessed to maintain consistency of the data structure. Locking the CEL does not ensure that events will be executed in chronological order. Additional mechanisms that synchronize the execution of processors or make use of model specific information are necessary to execute

multiple events concurrently. Further, the performance of such a system could be quite poor due to the contention for access to the CEL.

Specialized parallel discrete event simulation (PDES) algorithms have been developed to solve the problem of parallel and distributed synchronization. These algorithms employ multiple event lists, as opposed to CEL-based algorithms which employ a single event list.

Fujimoto [12] describes the *local causality constraint* as follows:

> A discrete-event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging timestamped messages obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order.

The above condition is sufficient but not necessary to guarantee that no causality errors occur. If two events are independent then it is not necessary to execute these events in timestamp order. In a parallel simulation a synchronization mechanism is required to ensure causal correctness for concurrently executed events.

3.1.3 Risk and Aggression

Reynolds [30] introduced the terms *risk* and *aggression* to help classify the different synchronization algorithms. Aggression is a property of synchronization algorithms that execute events before causal ordering is guaranteed. Risk is a property of synchronization algorithms that dispatch events before the causal order of the generating event is guaranteed.

Conservative synchronization algorithms strictly avoid causality errors by adhering to the local causality constraint; they do not exhibit risk or aggression. *Optimistic* algorithms exhibit aggression and risk to varying degrees; this requires that they detect and recover from any causality errors that might occur. *Aggressive no risk* algorithms execute events before their causal ordering is guaranteed, but do not dispatch

Algorithm class	Risk	Aggression	Algorithm
Conservative	X	X	CMB Null Message [3, 4] Deadlock detection [5] Cooperative Acceleration [1] Bounded Lag [25] Critical Channel Traversing [42]
Optimistic	√	√	Jefferson's Time Warp [19]
Aggressive No Risk	X	√	Bounded Lag (without Time Warp) [24] Breathing Time Buckets [37] SRADS/LR [9]

Table 3.1: Risk and Aggression of Synchronization Algorithms

new events until the causal order of the generating event is guaranteed. Table 3.1 classifies numerous PDES algorithms according the properties of risk and aggression.

3.2 Conservative Synchronization

Conservative synchronization implies that each LP in the model executes events in non-decreasing timestamp order. An event is *safe* to execute if the LP can determine that it is impossible to receive any event with a smaller timestamp. A safe event can be executed without violating the local causality constraint. An LP is said to be blocked if it has no safe events to execute. *Deadlock* refers to the situation in which all LPs are blocked. When all of the channels in a system are populated with events, LPs are unlikely to become blocked since there are always safe events to execute. However, deadlock can occur frequently if the ratio of unprocessed events to channels is too low, or if unprocessed events become clustered in one part of the simulation, i.e., if there are many empty channels.

There are two main classes of conservative synchronization, *asynchronous* and

synchronous. Asynchronous algorithms avoid synchronizing the execution of processors as much as possible, while synchronous repeated synchronize execution of the processors. Asynchronous algorithms must address the issue of deadlock either by avoiding it, or recovering from it when it occurs. Synchronous algorithms do not become deadlocked because they repeatedly stop the simulation and determine what events are safe to process.

Consider two LPs p and q. In order for p to affect q, it must send it a timestamped message (an event). For the following discussion, p is assumed to be the sending LP and q the receiver. p has lookahead ℓ with respect to q if p can be certain that it will not affect (schedule an event for) q for an amount of simulated time ℓ. The following categorization of lookahead is from Nicol [28]:

- *Bounded time* lookahead suggests that p knows that it will not affect q up to some time t. *Exact time* lookahead means that p knows exactly when it will next affect q, i.e., at time t.

- *Content* lookahead refers to the situation in which p has information about the content of the next message that it will send to q. With *time* lookahead (bounded or exact) only information regarding the time of future events is available. If p has both exact time lookahead and content lookahead, it is possible to immediately schedule the event for q.

- Lookahead can be classified as *Directed, Semi-directed, or Undirected* . This indicates whether the lookahead is with respect to a single LP (directed), a subset of the LPs (semi-directed), or all LPs (undirected).

- Some simulation algorithms, typically synchronous algorithms can exploit *conditional* lookahead. Conditional lookahead exists when p can say that it will not affect q before t, provided that some state condition of p does not change before time t. Lookahead that is not conditional can be referred to as *unconditional* lookahead.

Performance of conservative synchronization mechanisms is tied closely to lookahead. Conservative algorithms must be adept at determining what will not happen before a given point in simulation time. In general, a conservative synchronization mechanism cannot fully exploit the available parallelism in the model.

3.2.1 Chandy Misra Bryant (CMB) null message algorithm

Chandy and Misra [4], and independently Bryant [3] developed the first asynchronous conservative synchronization algorithms. These algorithms assume a static specification of channels. They also assume that events are sent down a channel in non-decreasing timestamp order, this allows an LP to determine a lower bound for the timestamp of the next event to arrive on that channel. The *channel clock* is defined to be the timestamp of the first event in the channel, or the timestamp of the last event received on that channel if it is currently empty. A lower bound for the timestamp of the next event to arrive on a channel can be computed by adding the lookahead between sending and receiving LPs to the clock of the sending LP.

A *zero lookahead cycle* refers to a cycle in the connection topology of LPs which contains zero lookahead, i.e., the lookahead of every channels in the cycles is zero.

Deadlock can be avoided through the use of *null messages* provided there are no zero lookahead cycles in the model. Null messages are timestamped events that have no associated action; they just serve to update the channel clock when they are processed. In the original CMB algorithm null messages are sent after each event is processed to inform neighboring LPs of a lower bound on the timestamp of the next possible event. The timestamp of a null message is set to be the local simulation time of the sending LP plus the lookahead between the sending and receiving LPs. Null messages do not allow for zero lookahead cycles to exist since there would still be potential for a deadlock situation in which the simulation time would not advance.

A *low lookahead cycle* refers to a cycle of LPs which contains inadequate lookahead to allow the execution of an event without first traversing the cycle to update clock

values. Low lookahead cycles can lead to large numbers of null messages being sent in the CMB algorithm.

Optimizations of the CMB null message algorithm involve reducing the number of null messages that must be sent. In particular cooperative acceleration [1] and carrier-null messages [40] attempt to reduce the number of null messages sent in low lookahead cycles. Another approach is to send null messages only on a demand basis [14]. This can be better or worse depending on how prone to deadlock the simulation is. *Implicit null messages* avoid sending null messages altogether by modifying the channel clock in place. This approach is applicable when the sending and receiving LPs are in the same memory space, as all LPs are on a shared memory parallel computer. Taking advantage of all available lookahead is important to obtaining good performance in an asynchronous conservative simulation [28] .

If a physical system is modeled using more LPs than there are processing elements available, this will require a scheduling mechanism to decide which LP should be executed next on a given processor. The CMB algorithm does not specify how LPs are scheduled. One simple approach is to select the next LP based on the current simulation times of all non-executing LPs in the model. Choosing the LP with the smallest simulation time is a reasonable heuristic since that LP is the furthest behind and has the greatest potential to advance.

3.2.2 Critical Channel Traversing (CCT)

The critical channel traversing algorithm (CCT) [42] is based on the CMB algorithm described above. The CCT algorithm attempts to determine which LPs are good candidates for execution by observing which LP is preventing the currently executing LP from continuing to execute. The goal of this approach is to maximize the number of events executed per LP execution session and reduce LP scheduling overheads.

Each LP in the CCT algorithm maintains a list of LPs that will be scheduled once its execution session terminates. The behaviour of the CCT algorithm can be

described in terms of a currently executing LP. Each LP is processed according to the following steps:

1. Determine the safetime; this is taken to be the minimum channel clock of all empty input channels.

2. Execute any events with timestamps less than or equal to the safetime, or until a previously non-empty channel becomes empty.

3. Update the safetime, if it has increased return to step 2.

4. There are no more events that are safe to execute, so determine which channel is preventing further execution. This is called the *critical channel.* The currently executing LP is added to the scheduling list of the LP connected to the critical channel.

5. Add any LPs in the local scheduling list to the central LP scheduling queue.

The TasKit implementation of CCT employs multi-level scheduling, instead of scheduling LPs, the CCT algorithm is used to schedule groups of LPs called *tasks.* Different types of tasks are then processed in different manners. LPs in a *cluster task* share the same event queue. The cluster task operates like a CEL simulation. *Pipe tasks* impose a fixed schedule on the execution order of LPs in the task. The tasks are taken from the centralized scheduling queue.

CCTKit is a second implementation of the CCT algorithm, however each processor has its own scheduling queue instead of a single centralized scheduling queue. CCTKit also uses tasks to provide multi-level scheduling.

3.2.3 Deadlock Detection & Recovery

Deadlock detection and recovery is an alternative to deadlock avoidance. Deadlock detection and recovery is again a conservative asynchronous approach to synchronization. This technique succeeds in eliminating null message traffic and it also allows for

zero lookahead cycles. Unfortunately in practice the performance is not good. The problem is that a large proportion of the simulation may become deadlocked prior to the entire simulation becoming deadlocked, this implies that only a small part of the potential parallelism can exploited. Chandy and Misra presented an algorithm based on detection and recovery in [5].

Fujimoto presents a version of the Dijkstra/Scholton algorithm for deadlock detection [14]. This algorithm assumes that there is a "controller LP" capable of recognizing a deadlock situation. A LP based simulation is an example of a diffusing computation in which individual LPs are initially blocked and only begin to execute once they receive an event. They execute for a period of time, possibly generating other events and sending them to other LPs until they stop. A stopped LP does not resume execution until it receives a message. The basic idea of this algorithm is to use a tree to maintain information regarding the LPs currently engaged in the simulation. When a message is sent to an LP, that LP adds itself to the tree if it is not already part of it. When an LP blocks, it will remove itself from the tree. When the controller LP becomes a leaf node in the tree it knows that the simulation has deadlocked.

Given that all LPs are blocked, it is always safe to process the event in the system with the smallest timestamp. The controller LP broadcasts a message asking each LP to report their minimum timestamped event. Once the controller has this information it determines the events which are safe to process and instructs the corresponding LPs to begin executing.

3.2.4 Synchronous Simulation Protocol

A synchronous simulation protocol cycles through two phases of execution. During the *global synchronization* phase, each LP is able to determine which events are safe to process. This is accomplished via a *barrier synchronization* operation which waits for all LPs to stop executing. During the *event processing* phase each LP executes all

of its "safe" events.

Starting and stopping the LPs has an advantage over certain asynchronous protocols. A simulation based on a deadlock detection and recovery algorithm, such as the one presented in the previous section, can degenerate into a sequential computation if it takes a long time for the deadlock to finally occur. It is possible that most of the simulation is deadlocked while a small portion continues to make progress. A situation like this can drastically degrade simulator performance[1]. On the other hand, a synchronous simulation protocol has more control over the amount of computation performed during each cycle of execution.

A barrier is a construct for parallel programming that can be used to synchronize the execution of multiple processors. Three different types of barriers are presented in [14]: centralized barriers, tree barriers, and butterfly barriers. Centralized barriers require a central controller processor and the technique does not scale well to a large number of processors. Tree barriers solve the scalability issue but still require broadcasting a message to indicate when global synchronization has been achieved. The butterfly barrier is good because there is no need for a central controller. After each processor has performed $\log_2 N$ pairwise barrier synchronizations, global synchronization has been achieved.

Transient messages are messages between processors that have been sent but have yet to be received. In order to support asynchronous messaging, care must be taken to observe all transient messages prior to determining which events are safe to process. Barrier synchronization is best suited to shared memory machines because of the communication overhead and the need to observe transient messages.

Bounded Lag [24, 25] uses a time window approach to reduce the amount of computation required to determine which events are safe to process. The trick with bounded lag is to determine a window size that is optimal. A window that is too small will result in few events being executed in each execution phase; if the window

[1] Amdahl's law: No more than k-fold speedup is possible if $\frac{1}{k}$th of the computation is sequential.

is too large the benefit of the window is lost, since the algorithm must again check the messages of all other LPs. Bounded lag is an example of an algorithm that can exploit both directed and undirected lookahead.

3.3 Optimistic Synchronization

Unlike conservative algorithms, optimistic synchronization techniques often process an event before it is known to be safe. This occasionally results in causality errors which must be detected by the simulation kernel and recovered from. The first subsection is focused on the Time Warp synchronization mechanism based on the *virtual time* paradigm. The second subsection briefly covers aggressive no risk algorithms.

Jefferson introduced the virtual time paradigm [19] as a method for organizing distributed systems. Virtual time is less restrictive than real time, it can progress forward and backward. Thinking of time in this way can be useful for many distributed systems. For example, database concurrency control. Time Warp [19] is an implementation of virtual time well suited to parallel discrete event simulation (PDES).

3.3.1 Time Warp for PDES

In Time Warp, each LP processes its events in non-decreasing timestamp order. However, if an LP receives a message in its past (a *straggler message*), it detects this violation of the local causality constraint and takes appropriate action to recover.

Rollback is the mechanism responsible for restoring the state of the LP so that the straggler message can be correctly inserted and executed. *Local virtual time* (LVT) refers to the current simulation time of an LP. Recall that at a given wall-clock time, each LP in the system may have advanced to a different point in simulation time. Local virtual time tends forward but it may occasionally jump backward when a rollback occurs. To correctly process a straggler message, local virtual time must be

rolled back to the timestamp of that message.

Each event processed with timestamp greater than that of the straggler message may have modified state and caused new messages to be generated. This implies that not only local state must be restored, but that certain messages may need to be "unsent". After the rollback is completed, the simulation can advance forward again, possibly re-executing the events that were just rolled back.

Messages in Time Warp are always created in pairs. The pair consists of a negative and *positive* copy of the information. Negative messages called *antimessages* are used to "unsend" previously sent positive messages. If antimessages and positive messages meet before either is processed then they cancel each other out and there is no record of either message ever existing. A message can be unsent by sending its corresponding antimessage.

When an LP receives an antimessage several things can happen. If the corresponding positive message has yet to be processed, then the antimessage and the positive cancel each other out. If the positive message has already been processed then the LVT has already advanced and the antimessage will cause the receiving LP to rollback. If an antimessage arrives before the corresponding positive message it can be ignored; when the positive message arrives they will cancel each other out.

Jefferson compares virtual time to virtual memory [19], many of the concepts in one have analogues in the other. Each time an LP processes an event it will discover that its timestamp is greater than the LVT (akin to a page hit in a virtual memory system), or that the event's timestamp is in the past with respect to LVT and that a rollback is required (similar to a page fault in a virtual memory system). This is the notion of a *timefault.*

The rollback mechanism relies primarily on LPs being able to restore their state. There are several different approaches to state saving:

- *Copy* state saving saves the entire state of the LP after each event is processed. This is the simplest mechanism to implement. However, it is very memory

intensive. The primary advantage of this technique is that restoring the state is quick.

- *Infrequent* state saving requires less memory than copy state saving by only saving the state of the LP periodically (e.g., after every 10 events). The problem with this approach is that the simulation may have to roll back further than logically necessary to reinsert the straggler message, and a greater number of events will need to be reprocessed.

- *Incremental* state saving records the changes to an LP's state after each event. This contrasts with the other approaches to state saving that save the entire LP state. This approach is more complicated to implement and restoring the state requires tracing backward through the saved state changes. The advantage of this approach can be a substantial savings in the memory requirements. However, it is possible that incremental state saving requires more memory than infrequent state saving in some situations.

A global control mechanism is required to ensure that a simulation does not run out of memory and to decide when it is safe to commit irrevocable operations such as I/O.

Global virtual time (GVT) is defined to be the minimum timestamp of all unprocessed messages in the system and those currently being executed, this includes both positive messages and antimessages. Care must be taken to observe any messages that are still in transit. GVT always progresses forward with respect to wall-clock time.

Fossil collection is the process of reclaiming memory when it is no longer needed by the Time Warp system. Once GVT has advanced beyond time t, any messages with timestamps less than t can be discarded. Any saved state that is no longer necessary to reconstruct the LP state at time t can also be freed. Similarly, I/O operations can only be committed once GVT advances beyond the simulation time

of those operations.

There are three GVT algorithms presented in [14]. The synchronous GVT algorithm, Samadi's GVT algorithm and Mattern's GVT algorithm. All algorithms must account for transient messages in the system. The synchronous GVT algorithm requires that all processors stop processing events while the GVT is being calculated. Samadi and Mattern's solutions allow processors to continue executing, but they must now account for the fact that different processors will report LVT at different points in wall-clock time.

The Time Warp mechanism does not rely on channels. Conservative synchronization mechanisms tend to impose artificial dependencies. For example, even though events are independent they must be executed in timestamp order. The Time Warp mechanism should perform well if the simulation obeys the temporal locality principal. That is, that most messages arrive in the future and those that arrive in the past, arrive mostly in the near past.

A rollback echo refers to the situation where the LVT is rolled back by an increasing number of simulation time units each time a rollback occurs. This situation can arise when rolling back the simulation by a certain amount of simulation time requires more wall-clock time than executing forward by the same amount of simulation time. Rollback echoes will continually slow the progress of GVT with respect to wall-clock time as the simulation progresses.

3.3.2 Aggressive No Risk

The *event horizon* is defined to be the difference in timestamp between the first event that is processed, and the last event that can be processed independently of any events generated in that cycle. Steinman [37] has explained how exploiting the concept of an event-horizon allows synchronization protocols to reduce the amount of risk when sending messages. One example of this strategy can be found in breathing Time Warp [36].

Aggressive no risk algorithms (ANR) must support a rollback mechanism but they do not need anti-messaging since messages are sent only once it is known that they will not cause rollbacks. By not propagating incorrect information ANR algorithms avoid compounding problems that occur in optimistic algorithms that exhibit both risk and aggression. For example, the likelihood of rollback echoes is less with ANR algorithms. There is an optimistic version of SRADS that allows for local rollbacks and aggressive processing of events [9].

3.4 Summary

This chapter has presented the necessary background in PDES. Several channel based conservative simulation algorithms were described. The CMB null message algorithm and the critical channel traversing algorithm will be implemented in the test system described in Section 4.2.1. Their sequential performance will be evaluated in comparison to the sequential simulation algorithms described in the previous chapter using a synthetic workload model described in Section 4.1.7. Several other conservative, optimistic, and aggressive no risk synchronization mechanisms were also discussed. Optimistic algorithms which exhibit risk or aggression are not good candidates for sequential execution. The overheads of state saving, rollbacks, etc., are prohibitive in a sequential execution environment.

Chapter 4

Model and Methodology

The performance of simulation algorithms is usually compared according to some simulation model. The question becomes which algorithm can simulate the selected model in the least amount of time. Of course some algorithms are better suited to some models. The algorithm that performs best for one model might not be the best algorithm in another situation. Synthetic workload models are one way that the range of performance of an algorithm can be explored.

The test system used in this thesis is based on a synthetic workload model. The test system allows the comparison of a suite of simulation algorithms under many different conditions. All simulation algorithms execute the same model level code which helps to ensure a fair comparison of the algorithms. In the next chapter experiments are conducted that vary parameters of the synthetic workload. Algorithm performance is reported in terms of six metrics that are defined later in this chapter.

Another way that algorithms can be compared is according to their asymptotic complexity or behaviour. This does not give a complete picture of algorithm performance on its own, since it does not reflect the cache behaviour of the algorithms. However, it does give an indication of how and when the channel based conservative algorithms can achieve better performance than CEL-based approaches.

This chapter begins with a survey of existing synthetic workload models, then proceeds to detail the model that is used for experiments in this thesis. The second section describes the test system and experimental methodology used in comparing the sequential execution of the various simulation algorithms. The final section of this chapter analyzes the complexity of sequentially executed channel based conservative synchronization mechanisms, in general and for the workload model used in this thesis.

4.1 Synthetic Workload Model

Synthetic workload models attempt to capture the essential characteristics of a wide variety of simulations, rather than any particular real world system. The relative simplicity of a synthetic workload model makes it easier to understand the behaviour of the simulation algorithms without having to deal with the specifics of a real world simulation. Synthetic workload models allow stress testing of algorithms under a variety of conditions. This helps in understanding the limits of performance for a particular algorithm. Assuming that a synthetic workload model accurately captures the characteristics of a real simulation, results from the synthetic model can be used to estimate performance of the algorithm in a real world simulator. Subsections 4.1.1 through 4.1.6 describe different synthetic workload models that have been used to test simulation algorithms. Subsection 4.1.7 explains the workload model used in the experiments conducted in this thesis research.

4.1.1 HOLD model

Many comparisons of CEL algorithms have been based on the HOLD model [20, 2, 32, 31, 39, 27]. The model gets its name from the Simula[1] hold operation. In the context of this thesis, a hold operation consists of removing the event with the

[1]Simula was a programming language with mechanisms to support development of simulations.

Distribution	HOLD model	PHOLD	Test
Constant	N/A	1.0	1.0
Exponential	$-ln(rand)$	$0.1 - ln(rand)$	$-ln(rand)$
Uniform	$2rand$	$0.1 + rand$	$2rand$
Biased	$0.9 + 0.2rand$		
Bi-model	$0.95238rand+$ if $rand < 0.1$ then 9.5328 else 0		
Triangular	$1.5\sqrt{rand}$	N/A	$1.5\sqrt{rand}$

Table 4.1: Scheduling Distributions

smallest timestamp from the pending event set, incrementing its timestamp by some amount and then reinserting it. A complete description and analysis of this model can be found in [27].

A hold operation can be simulated by any central event list algorithm using a remove_min operation followed by an insert operation. Each simulated hold operation removes an event with timestamp t and results in the generation of a new event with timestamp $t + \Delta t$ where Δt is a random variate taken from a distribution $F(t)$. The HOLD model column of Table 4.1 shows distributions that are commonly used with the HOLD model in evaluating CEL-based algorithms.

The model parameters are P and $F(t)$. P is the size of the pending event set and $F(t)$ is the distribution used to determine when events will occur. F is known as the scheduling distribution. The HOLD model has three phases:

1. [INITIALIZATION]. P events are inserted into the pending event set, timestamps of the events are generated from the distribution $F(t)$.

2. [TRANSIENT]. Execution of M_1 hold operations to allow model to reach a steady state.

3. [STEADY STATE]. Execution of M_2 hold operations while measuring CEL

performance. Performance can be measured in terms of time elapsed during execution of M_2 hold operations or the number of operations required to perform M_2 hold operations.

Based on analysis of the interaction HOLD model (see Section 4.1.2), McCormack and Sargent [27] determined that the HOLD model should favour scheduling distributions with a coefficient of variation greater than 1, such as the hyper-exponential or mixtures of exponentials. When more than one scheduling distribution interact as is the case for modeling most real world situations the coefficient of variation tends to be greater than 1. This suggests that the scheduling distribution should also be chosen to have a coefficient of variation greater than 1 to be more realistic.

There are three potentially important characteristics of real world simulations that the HOLD model fails to address. The first issue is that the event population is fixed. The pending event set remains a constant size once the INITIALIZATION phase has completed. There are many real world scenarios where the number of events is likely to change throughout the simulation. A second issue is that the timestamps of the entire event population are generated from the same scheduling distribution. Even very simple real world simulations usually involve multiple scheduling distributions. Choosing a scheduling distribution with a coefficient of variation greater than 1 will help address this concern, but it is not a complete solution. The third issue is that all of the events in the HOLD model are independent of one another, this is never the case in a real world simulation. In a real world simulation the generation of new events depends on the current system state, which was previously acted upon by other events. Despite these shortcomings, the HOLD model has been used in many empirical comparisons of CEL-based algorithms.

4.1.2 Interaction HOLD Model

In a real world simulation, timestamp increments are likely to be taken from multiple distributions. For example, a Poisson distribution is used to model the arrival of

customers while a normal distribution is used to model the service times for those customers. This interaction of distributions is not captured by the HOLD model. McCormack and Sargent [27] propose the interaction HOLD model that addresses the situation where different scheduling distributions interact. The interaction HOLD model is identical to the HOLD model except that event timestamps are generated using multiple distributions $F_i(t)$. The pending event set contains P independent events; each hold operation removes an event E_i, increments its timestamp by Δt where Δt is taken from a scheduling distribution $F_i(t)$ and then reinserts the event into the pending event set.

4.1.3 Up and Down HOLD Model

Randy Brown [2] presents another variation of the HOLD model that compares algorithm performance for queues that change in size over time. The original HOLD model assumes that the pending event set size remains fixed. The Up and Down variation of the HOLD model measures performance as the queue size changes over time. Using a random sequence of enqueues and dequeues the queue is first grown until it reaches P elements and then emptied. During the growth phase enqueues occur with probability $p \in (0.5, 1.0]$, dequeues with probability $1 - p$. While the queue is being emptied the probabilities are reversed. The parameter p effectively controls the rate of queue size change.

4.1.4 Dependent HOLD model

Marín [26] introduces another HOLD model variation that addresses the event dependence issue. Marín's model requires that the CEL algorithm support an operation for deleting an arbitrary element in the collection. This synthetic workload model views the system as P objects that schedule events independently. The interactions occur randomly according to a probability parameter. The higher the probability the greater the frequency of interaction. If the probability is 0 then there is no interac-

tion. If the probability is 1 then after each hold operation, the object interacts with another randomly chosen object. When an interaction occurs, an object is randomly chosen and the corresponding event is deleted. A new event is created based on the last executed hold operation and then inserted into the pending event set.

4.1.5 PHOLD model

Fujimoto introduced a parallel HOLD (PHOLD) model used to compare the performance of conservative simulation algorithms based on deadlock avoidance and deadlock detection and recovery [11]. This synthetic workload model addresses the spatial characteristics of a simulation [11]. This is particularly important for evaluating parallel simulation algorithms since the spatial characteristics directly affect the available parallelism. A later version of the PHOLD model described in [13] was used to evaluate the performance of the optimistic TimeWarp PDES algorithm.

Recall the LP modeling methodology described in Section 3.1.1. The PHOLD model consists of a set of LPs connected in a toroid network as shown in Figure 4.1. When a event arrives at an LP, a busy wait loop is executed to simulate the computation grain associated with processing the event. The timestamp of the event is updated, and it is sent to a neighboring LP. As events move between LPs their timestamps are incremented to model the passage of time.

There are several parameters for this synthetic workload: timestamp increment, event population, topology of logical processes, routing probability and computation grain. The timestamp increment parameter selects one of the distributions from the PHOLD column in Table 4.1. In order to use the PHOLD model to compare conservative algorithms, the uniform and exponential distributions must be shifted. This guarantees that the lookahead is strictly greater than 0, a necessary condition for the deadlock avoidance algorithm. A pseudo random number taken from the specified distribution determines the amount of simulation time that an event is delayed as it travels from one LP to the next. This parameter corresponds to the scheduling distri-

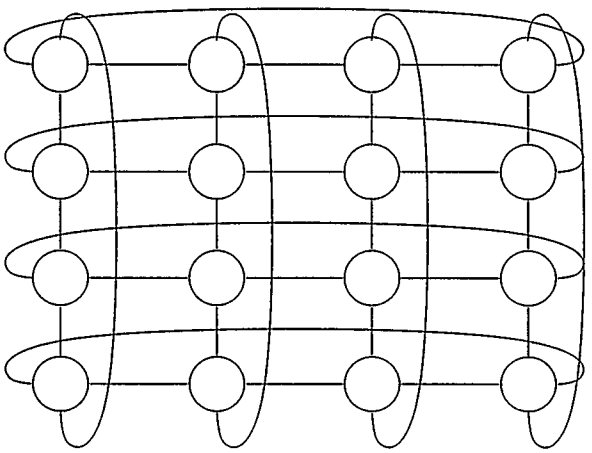
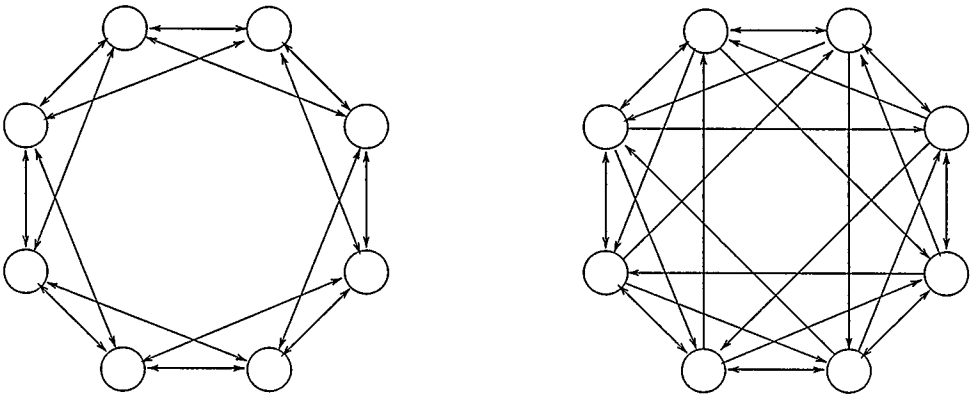


Figure 4.1: 4x4 Toroid Network

bution ($F(t)$) used in the HOLD model. The event population parameter determines the number of events in the system in the same manner as the P parameter for the HOLD model. The number of events remains fixed throughout the simulation. The topology of LPs can be arbitrarily specified. The original PHOLD workload model used toroid networks of 4x4 and 8x8 LPs. When an event is finished processing, it is forwarded on a randomly chosen output channel. The likelihood of sending on a particular channel depends on the routing probability parameters. Finally, the computation grain parameter is used to control the amount of time spent in the busy wait loop, this can be stochastic or fixed. Modeling computation grain is necessary in the parallel simulation setting since the amount of time that a cpu is busy will affect when it can send and receive events and ultimately simulator performance.

4.1.6 Ring Model

Liu and Nicol [23] developed a synthetic workload model that can be used to stress test the LP scheduling mechanisms of channel based conservative PDES algorithms. This model consists of an ordered set of N LPs connected in a ring. Each LP is connected to R of its predecessors and R of its successors as shown in Figure 4.2. The



a) Ring Topology b)Star Topology

Figure 4.2: Test Model Topologies, 8 LPs with Connection Radius 2

connections are symmetric and never duplicated. If there is a channel from LP A to LP B, then there is also a channel from LP B to LP A. There will never be more than one channel from LP A to LP B. The resulting topology is referred to as the *ring topology*, the parameter R is called the connection radius. In the Ring model, no events sent between LPs. Events are only scheduled locally within an LP. At any given time there will be at most one event in each LP's local event list. When a local event is executed it causes a new local event to be scheduled using an exponentially distributed timestamp increment. The exponential distribution has a mean of 1/D, where the D parameter refers to event density. There are approximately D events executed by each LP per second of simulated time. Again, there are no events sent between LPs. The reason for this was the motivation to focus on the overhead of the synchronization protocols.

4.1.7 Test Model

This subsection describes the synthetic workload model used for the experiments presented in this thesis. Any future reference to the *test model* can be taken to refer to the model described in this section. The test model incorporates many of the

parameters and characteristics of PHOLD and Ring models previously described. The test model was designed to be descriptive, yet simple to analyze. The parameters of this model are: number of LPs (N), event density (D), connection radius (R), channel delta (L), topology (T), computation grain (G), LP state size (S), and scheduling distribution ($F(t)$).

The simulation is initialized with $N \times D$ events. These events are uniformly distributed among all LPs in the system such that on average there are D events scheduled locally at each LP. The timestamps of these initial events are taken from the scheduling distribution $F(t)$. The event population remains constant throughout the simulation.

There are two connection topologies implemented in this workload model, a ring topology identical to that described in Section 4.1.6 and a *star topology*. Consider an ordered set of N LPs. In the star connection topology each LP is connected to its immediate successor and predecessor, and then connected to $2R - 2$ LPs spaced approximately equidistantly around the ring. The connections in the star topology are not necessarily symmetric. The existence of a channel from LP A to LP B does not imply the presence of a channel from LP B to LP A. There will never be more than one channel from LP A to LP B. Figure 4.2 shows a sample of the star connection topology with 8 LPs and connection radius $R = 2$. Each LP will have exactly $2R$ input channels and $2R$ output channels. In addition, each channel has an associated channel delay that reflects the minimum lookahead L, between the source and destination LP.

Given an LP and E, the next event to be processed, the following rules describe the behaviour of the LP. If E was generated locally, then send a new event to a neighboring LP via a randomly selected output channel. There is a uniform probability of selecting any particular channel. The timestamp of the new event is the timestamp of E plus the channel delta L. If E has arrived from a neighboring LP, schedule an event locally for time $T(E) + \Delta t_i$, where the Δt_i timestamp increment is taken from the scheduling

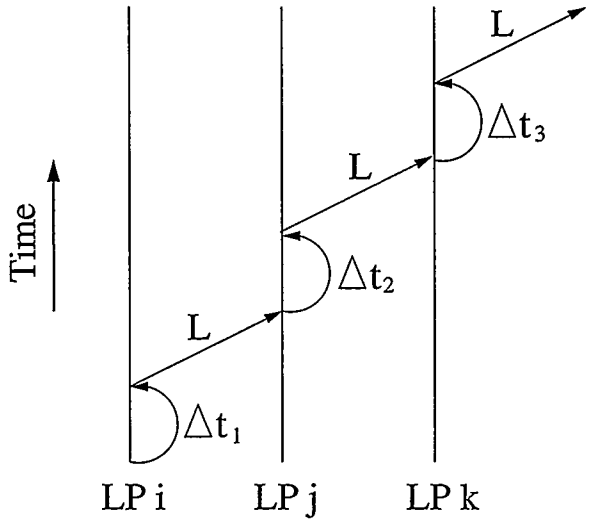


Figure 4.3: Event Time Line

distribution $F(t)$. Figure 4.3 shows the timeline of an event in the system. In the diagram, L is a constant delay which corresponds to the channel delta. Δt_1, Δt_2, Δt_3, ... are timestamp increments taken from a stream of pseudo random numbers that fit the distribution $F(t)$.

The test model parameters are summarized below.

1. N - The number of LPs parameter impacts the number of events in the system since the event population is equal to the product of the number of LPs and the event density. More importantly, manipulating the number of LPs will affect the spatial characteristics of the simulation; with a larger number of LPs, there is greater parallelism available in the model.

2. D - The event density parameter also affects the number of events that are instantiated at the beginning of the simulation. $N \times D$ events are created system wide and then uniformly distributed among the LPs. On average D local events are scheduled for each LP. The timestamps of these events are taken from a distribution $F(t)$ with a mean of 1.0.

3. T - The model topology parameter allows simulation of either a ring or star connection topology as previously described.

4. R - The Connection radius parameter controls the number of neighbors that each LP is connected to. Each LP has $2R$ output channels on which it may send events, and $2R$ input channels on which it may receive events. Connection radius is used to model the topological characteristics of a system.

5. L - The channel delta parameter affects the available lookahead in the model. When the channel delta is small, LPs can quickly affect one another since there is little delay between them. When the channel delta is larger, LPs behave more independently since they can be further separated in time. Channel delta can be used to model the physical separation of the interacting processes. Channel delta, like the number of LPs, helps to model the spatial characteristics of the a system.

6. G - The computation grain parameter controls the amount of time required to process an event. It specifies the average time that a busy wait loop is executed in order to simulate the cost of processing an event.

7. S - The LP state size parameter models the additional LP state that would be accessed in a real simulation. For example, in a network simulation where each LP models a network router, additional state might be required to model output buffers and routing tables.

8. $F(t)$ - The scheduling distribution selects one of the timestamp increment distributions from the Test column in Table 4.1. All scheduling distributions have the same mean of 1.0.

To summarize the test model in terms of the previous models, it is similar to the PHOLD model, however the topology of the LPs bares more resemblance to the ring model. There are also differences in LP behaviour since test model uses both local

and external events. The PHOLD model focused on external events and the Ring model had only a single local event per LP.

4.2 Experimental Methodology

This section describes the experimental methodology. It is split into several subsections which describe various aspects of the methodology. The test system is described in Section 4.2.1. The implementation details are described in sections 4.2.2. The test architecture is described in Section 4.2.3. Section 4.2.4 describes the performance metrics that are used to compare the simulation algorithms.

4.2.1 Test System

This section describes the test system used to perform all experiments conducted in this thesis. The test system simulates the test model described in Section 4.1.7. In addition to the parameters described for the test model, the test system has parameters which control the length of a simulation run and the simulation kernel used to execute the simulation. The length of the simulation can be specified in terms of simulation time or wall-clock time using the `-sim_end_time` or `-wall_end_time` parameters. The simulation kernel is selected using the `-alg` and `-pq_impl` parameters. For the CEL-based algorithms these parameters allow the selection of the CEL implementation. For conservative algorithms, these parameters select the LP scheduling mechanism and the local event queue implementation. The test system implements many of the CEL algorithms described in Section 2.2 and several variations of the channel based conservative PDES algorithms described in Section 3.2. These additional test system parameters are summarized below.

1. `-sim_end_time` This parameter specifies a simulation end time. Once the simulation reaches this simulation time, it is terminated.

2. `-wall_end_time` This parameter terminates the simulation after a specified amount of wall-clock time has elapsed.

3. `-alg` This parameter is used to specify either CEL, or one of the LP scheduling mechanisms: PQ, CCT, or FIXED. The CEL option executes the well known central event list algorithm. PQ, CCT, and FIXED options execute variants of the CMB algorithm. The PQ option schedules LPs for execution according to their local time. The CCT option executes the critical channel traversing algorithm described in Section 3.2.2. Both the PQ and CCT options make use of a LP scheduling queue implemented using a heap that is optimized for hold operations. The FIXED option schedules LPs according to a fixed ordering using a FIFO queue.

4. `-pq_impl` In the case of the CEL algorithm, this parameter specifies the implementation of the central event list. For conservative algorithms it is used to specify the implementation of the local event queue. Options for this parameter are: list, henriksen, heap, splay, and calendar. These options correspond to the linked list, Henriksen's, heap, splay tree, and calendar queue priority queue implementations described in Section 2.2. The heap option selects a heap implementation that is optimized for executing hold operations.

Preliminary tests were performed using different combinations of LP scheduling mechanism and local event queue implementations. Three implementations were identified as good candidates for the local event queue. These were the list, heap, and splay options mentioned above. Because most experiments did not use large event densities, the implementation of the local event queue did not have a significant effect. As such, only results for kernels using the linked list local event queue implementation have been presented. There is one exception to this in Section 5.1.3 where the heap was used to implement the local event queue. This was necessary due to the large event densities encountered in that experiment.

All experiments were conducted using each of the PQ, CCT, and FIXED LP scheduling mechanisms and either the list or heap local event queue implementation. The behaviour of the conservative PQ and CCT algorithms was very similar, so results are only presented for the CCT algorithm. The choice to present results for CCT rather than PQ was motivated by the desire to understand the good sequential performance of the CCT algorithm observed in ATM-TN [41] and IP-TN [21]. Results for the PQ LP scheduling mechanism are presented in [6].

Four CEL implementations were selected for comparison with conservative algorithms. All experiments were conducted using each of the henriksen, heap, splay, and calendar CEL implementations. These algorithms demonstrate the range of possible performance for CEL-based approaches. There are some model parameters that have very little affect on the behaviour of CEL-based algorithms. In these experiments the results are only presented for the calendar queue as it was the best performing of the CEL-based algorithms.

4.2.2 Implementation and Memory Management

While comparing the different simulation algorithms, it was observed that small differences in the implementation can have significant impacts on performance. Two examples of this are described here.

A priority queue of events can be implemented using placeholders which each contain a pointer to an event or alternatively, the priority queue can be implemented so that it operates directly on the events themselves. Using placeholders allows a priority queue to be used more generally, but it introduces a level of reference that is unnecessary. In DES, this level of reference can be eliminated by adding additional data members to the event objects. For example, heap performance was improved by 10% by adding left and right child pointers to the events and avoiding the additional level of reference.

Simplifying the logic used in a priority queue by avoiding unnecessary tests can

greatly improve performance. For example, eliminating one conditional in the heap code improved performance by about 10%. Certain assumptions made by some of the specialized priority queue algorithms may have allowed them to avoid testing certain conditions. For example, Henriksen's algorithm assumed that the queue would never be empty or that the user would monitor the queue size independently of the algorithm. Eliminating the check for an empty queue is fine for the HOLD model tests since the queue is never emptied. For a general purpose simulator, the algorithm would need to check if the queue is empty before attempting to remove an event. Adding this check is likely to negatively impact performance.

One important aspect of implementing an efficient simulation kernel is to provide a specialized set of memory management routines. In a simulation, events are created and destroyed millions of times per second. Memory management using C++ operators new and delete, or C functions malloc and free can incur substantial overhead. These routines were designed to be general purpose; they can handle objects of varying size. In a simulation, events can usually be assumed to be of a constant size. If there are many event types, the size is taken to be the maximum of all event sizes. Event records are the primary concern rather than the auxiliary data which can be managed separately. A specialized memory allocation scheme can be designed that is very efficient for allocating fixed size chunks of memory, such as events.

A second aspect that is particularly important to parallel simulation is memory alignment of events and simulation objects. Memory alignment has numerous benefits. Assume that the size of an event is smaller than the length of the cache line. If the event falls entirely within a single cache line, then it can be retrieved from memory in one go. However if it crosses a cache line boundary, it will require two memory accesses. This is very expensive. Roughly speaking, a memory access is two orders of magnitude more expensive than a cache hit. Also, it may fill up the cache faster since it will require two cache lines and result in more cache invalidation. In parallel there is an added cost that occurs when multiple objects share the same cache line.

If one of the objects is modified, this invalidates the entire cache line and it means that every processor caching an object sharing the same cache line will need to reload that cache line.

Cache performance can be improved by ensuring that objects are allocated on cache line boundaries and that no objects share the same cache line. Allocating objects on cache line boundaries ensures that the object will occupy the fewest possible cache lines. If the object is not aligned to the beginning of a cache line it will use more cache memory and require additional reads and writes to update. Padding objects so that they do not share cache lines ensures that modification of one object cannot invalidate the cached copy of another. This is particularly important in parallel execution where cache lines might be accessed by multiple processors and result in premature invalidation.

Cache alignment can require more memory so in some situations it can have a negative affect on cache performance. For example, if cache aligning objects means that only half as many objects will fit in the cache, then the benefit of cache alignment may be offset by fewer objects being cached at any given time.

The test system uses specially designed memory management routines optimized for allocating and deallocating events. Other objects are allocated using the standard C++ new operator. In addition, the test system uses cache aligned events and cache aligned LPs. Each event is cache aligned and padded to occupy the entire cache line. In the test environment each event occupies exactly a single cache line due to its size being equal to the length of one cache line. Each LP is cache aligned and padded to the next larger multiple of cache line length. Since the size of an LP is not a multiple of cache line length some memory is wasted in cache alignment of LPs. It should also be noted that although the LPs themselves were cache aligned, dynamically allocated data members of the LPs were not. This was an oversight at the time of implementation and it could have some affect on performance.

In some situations, particularly for models with a large number of LPs it was

observed that cache alignment of LPs could have a slight negative impact. One explanation for this is the larger memory requirement to cache align the LPs. Cache aligning each LP requires more space and potentially results in more cache misses. Cache alignment of events was observed to improve performance by as much as 10%. This improvement is due to avoiding unnecessary cache invalidation that would occur if an event were to overlap a cache line boundary. Cache alignment of events did not require additional memory since as mentioned previously, the event and cache line sizes were equal. Further work would be required to quantify the affects of cache alignment on simulator performance.

4.2.3 Test Architecture and Compiler

All experiments were conducted using a Dell desktop computer with an 866 MHz Pentium III processor and 128 MB of RAM. This computer has a 16KB, 4-way set associative first level (L1) instruction cache with a 32 byte line size, a 16KB, 4-way set associative first level (L1) data cache with a 32 byte line size, and a 256KB, 8-way set associative second level (L2) cache with a 32 byte line size. The operating system was Red Hat Linux 7.3 with the v2.4.18 kernel.

There was some dependence observed on the optimization level used in compilation. In one situation the performance order of the algorithms actually changed depending on which level of compilation was used: "-O0", "-O1" and "-O2". Optimizations tended to reduce the observed performance difference among the algorithms. It would be interesting to examine what particular optimizations benefited the individual algorithms, but this is beyond the scope of this thesis.

The results for all experiments conducted in this thesis were obtained using software that was compiled with level 2 optimizations enabled (using the -O2 flag) using GNU g++ version V2.96.

4.2.4 Performance Metrics

In order to compare the different algorithms, six performance metrics have been defined. The majority of these metrics are obtained from calculations based on the output of valgrind, an open-source memory debugger for x86-GNU/Linux. Cachegrind is one of the valgrind performance tools and is used to perform detailed simulations of the instruction and data caches, while executing the different simulation algorithms. It records cache performance data for each function and procedure in the program. Cachegrind reports level 1 and 2 cache misses. Level 2 cache misses have a greater impact on performance than level 1 cache misses so all cache metrics have been defined in terms of the number of level 2 cache misses (reads or writes). For *model level cache behaviour* and *kernel level cache behaviour* the metric is reported as a percentage of the total memory accesses. The *amortized aggregate cache behaviour* metric is reported as the total number of level 2 cache misses amortized over the number of events executed. Metrics obtained from cachegrind are based on a simulation run that lasted 100 seconds of simulation time. The bias of simulation initialization was removed by executing the initialization code separately and then subtracting these measurements from those obtained in the run of 100 seconds of simulation time. The event rate metric is obtained directly from the execution of the native code and is based on a simulation run that lasted 60 seconds of wall-clock time. There is no bias in the execution of the native code, since the 60 seconds of wall-clock time does not begin until initialization has completed.

The first metric is *model level cache behaviour*. The model level cache behaviour measures data cache behaviour in the `process` method and in any calls associated with random number generation. Recall that the simulation API requires the modeler to implement the process method for the logical processes and this is where the majority of the simulation specific work occurs. Random number generation is used in the process method when choosing a timestamp for the next local event.

The second metric also measures cache performance, but in the simulation algo-

rithm rather than in the model level code. *Kernel level cache behaviour* excludes memory accesses related to random number generation and the `process` method.

Amortized aggregate cache behaviour is a third cache metric that combines both the kernel and model level cache behaviour. This metric is amortized over the number of events, it is the average number of level 2 data cache misses that occur in processing a single event.

For PDES algorithms, *events per LP execution* is defined to be the average number of events that are processed each time an LP is executed. It is computed by taking the total number of events executed during the simulation and dividing by the total number of LP execution sessions. Although events per LP execution is model dependent, it is still helpful in explaining algorithm behaviour and performance. In a conservative parallel simulation, an events per LP execution less than one would suggest the presence of a low-lookahead cycle that might warrant re-partitioning the model. The concept of events per LP execution can be extended to CEL-based algorithms by counting the average number of consecutive events executed at one LP before executing an event at a different LP. Note that a CEL-based algorithm always achieves an Events per LP execution greater than or equal to 1; frequently it is very near 1 due to the total timestamp ordering imposed by a CEL-based algorithm.

The *events per LP execution* metric is extracted from the cachegrind run, but it is obtained directly from the simulation kernel. Due to the synthetic workload model that was used, the events per LP execution is approximately 1 for all CEL-based algorithms. There are workload models in which events are localized in time and space where the events per LP execution could exceed 1; however, this test model does not explore these situations. As such, the events per LP execution plots for CEL-based algorithms will not usually be commented upon. Attention is focused on the behaviour of the different conservative scheduling mechanisms. Also note that plots of events per LP execution have logarithmic scales for both the x and y axis.

The fourth metric is *kernel level amortized computation cost.* It is a count of all

instructions (excluding the model level code) divided by the number of events executed by the simulation kernel. The metric is obtained using data from the cachegrind tool (cachegrind also counts the instructions per function or procedure). The metric excludes instructions that are executed in the model level code, including random number generation. The decision to exclude model level code was motivated by the desire to highlight the differences between the algorithms.

The most comprehensive metric is *event rate.* It corresponds to the number of events executed per second of wall-clock time. This metric is obtained by executing the simulation for a specified amount of wall-clock time and then calculating the event rate based on the number of events that were executed. It should be noted that different implementations of the same model level behaviour can result in significant differences in event rate. Consider a network simulation that uses 3 events to model FIFO transmission of a packet through a router. As described in [7], the same behaviour can be accomplished using only 2 events. The event rates for these two model implementations could vary significantly, even though they model the same behaviour. The experiments in this thesis compare the algorithms under the same implementation of model level behaviour. The event rate metric can hide the variation between the algorithms since it includes the cost of the model level code associated with the simulation. For the same reason, it is the most realistic metric as it corresponds exactly to how someone would use the simulation in practice.

4.2.5 Experiment Outline and Parameters

A single simulation run was completed for each point in each plot. Multiple simulation runs could have been used to obtain confidence intervals, but due to the well behaved nature of the simulation metrics this was not done. 100 seconds of simulation time and 60 seconds of wall-clock time were both adequate to obtain a stable measurement for each point. For the results published in [6], 5 simulations runs were adequate to achieve a 95% confidence interval such that the half-width was within 5% of the

sample mean for every point.

In general, the controlled experiment parameters have been chosen to simplify the analysis and highlight the differences among the algorithms. For example, using the uniform distribution as the scheduling distribution demonstrates the greatest variation among the algorithms, so only these graphs are presented. Other experiment parameters have been chosen to simplify the analysis. For example, the connection radius might be set to 1 so that issues related to larger connection radii do not come into play.

4.3 Analysis

In this section the theoretical computation cost of the conservative simulation algorithms will be examined. The computation cost here refers to the number of kernel level instructions required to execute the simulation. First a general cost expression is described in terms of the events per LP execution metric. Then an estimate for the minimum and maximum number of events per LP execution are developed. Finally the asymptotic behaviour of channel based conservative algorithms is compared to that of CEL-based algorithms.

4.3.1 Asymptotic Bounds

Just as the asymptotic behaviour of CEL-based algorithms depends on the implementation of the priority queue, the behaviour of conservative algorithms depend on the implementation of their subcomponents. For a channel based conservative algorithm, the implementations of the LP scheduling queue, safetime calculation and local event queue will affect asymptotic performance.

The cost of executing a single event breaks down into three parts; the LP scheduling queue cost, the safe-time calculation cost and the local event queue cost. Consider a model with N LPs, an average event density of D events per LP, an average of C

channels per LP. Suppose that on average the simulation kernel executes E events per LP execution session. Table 4.2 summarizes the asymptotic bounds in terms of the different synchronization algorithms and different algorithm components.

The LP scheduling queue cost is associated with inserting or removing an LP from the LP scheduling queue. For the PQ and FIXED schedule algorithms, a single LP is removed from the LP scheduling queue and the same LP is returned to the queue once its execution session is complete. The behaviour of the CCT algorithm is more complex. A single LP is removed from the LP scheduling queue but after completing its execution it is not reinserted into the scheduling queue. Instead it schedules all of the neighboring LPs that had marked their connection to it marked as critical by inserting them into the LP scheduling queue. Still, the asymptotic LP scheduling queue cost for CCT is dominated by the cost of an insert or remove operation rather than the number of insert operations an LP performs per execution session.

The LP scheduling cost depends on the synchronization algorithm. If a heap is used to schedule LP execution and all LPs in the system are maintained in the LP scheduling queue, as is the case with the PQ algorithm, then the cost is $O(\log_2 N)$. Another approach is to execute each LP in the system according to some fixed ordering, this type of LP scheduling queue is used by the FIXED schedule algorithm and has a constant access cost $O(1)$. The critical channel traversing algorithm is more complicated. One of the advantages of the CCT algorithm is that it is not necessary to maintain all LPs in the LP scheduling queue. This results in a smaller queue and smaller access costs. The LP scheduling queue in CCT is implemented using a heap, so again the cost is logarithmic in N', the number of LPs in the scheduling queue. Empirical observations indicate for the test model used in thesis experiments, in the case of connection radius 2, ring topology, the LP scheduling queue contains on average $N/2$ LPs. Increasing the connections radius increases the number of LPs in the LP scheduling queue.

The cost of scheduling LPs is a function of the number of LPs and depends on

the choice of scheduling mechanism (FIXED schedule, PQ, CCT, etc.) and on the implementation of that scheduling mechanism.

The second part of the cost of executing an event is computing the safe time for an LP execution session. Computing the safetime requires determining the minimum possible timestamp of the next event that could arrive on each empty channel. For algorithms tested in this thesis all channels are examined regardless of whether they are empty. This is used to ensure that each channel that has events, has a representative in the local event queue. The cost of the safetime calculation is linear in the number of channels, $O(C)$.

The third part of the cost of executing an event is removing the event from the local event queue. The cost to obtain an event from the LP's local event queue is a function of the number of events in the queue. This cost depends on D, the average event density in the model and the implementation of the local event queue. If the local event queue is implemented using a heap the cost is $O(\log_2 D)$, while if implemented using a linked list the cost is $O(D)$. If no events are generated locally then the size of the local event queue is bounded by the number of channels since it contains at most one representative from each channel. However, the test model under consideration in this thesis uses local events so the cost depends on the average event density in the model D.

Both the LP scheduling queue cost and the safetime calculation costs are amortized over the execution of multiple events. The per event cost for these components is divided by E, the average number of events executed per LP execution session.

4.3.2 Events per LP execution

The next part of this analysis is to derive an expression for E, the number of events executed per LP execution session.

The *lifetime* of an event refers to the difference between its timestamp and the timestamp of the event that caused its generation. If μ is the average lifetime of an

Synchronization Algorithm	LP scheduling queue cost	Safe time calculation cost	Event queue cost	Per event cost
FIXED	1	C	$\log_2 D$	$\frac{1+C}{E} + \log_2 D$
PQ	$\log_2 N$	C	$\log_2 D$	$\frac{\log_2 N+C}{E} + \log_2 D$
CCT	$\log_2 N'$	C	$\log_2 D$	$\frac{\log_2 N'+C}{E} + \log_2 D$

Table 4.2: Asymptotic bounds for channel based conservative synchronization

event in the simulation then in one second of simulation a time, $1/\mu$ events can be executed. If there are D timelines within a given LP, then D/μ events are executed at that LP per second of simulation time. Suppose the available lookahead allows an LP to advance its clock by L seconds of simulation time, then the number of events executed during that LP execution session is LD/μ. On average there are $E = LD/\mu$ events executed per LP execution session.

In the case of the PQ algorithm, all LPs are stored in a priority queue according to their clock. The timestamp of an LP taken from the LP scheduling queue must have the smallest timestamp in the system. The timestamp of the LP will be less than or equal to the timestamps of each of its neighbours. This means that it can advance at least the minimum lookahead L.

A similar situation occurs for the FIXED schedule algorithm. Every LP in the fixed order is executed exactly once, prior to subsequent execution of LPs that have already executed. Each time an LP executes, its clock will be less than or equal to each of its neighbors. This means that every LP will advance at least the minimum lookahead L.

This is also true for the CCT algorithm. When an LP executes, it has the smallest timestamp of any LP in the scheduling queue. There may be other LPs that have yet to be scheduled placed in the scheduling queue, but their local time would be greater than or equal to the local time of the currently executing LP. By the same rational, each LP can advance by the minimum lookahead, L.

Synchronization Algorithm	Per event cost	Parameters			
		N	D	C	L
FIXED	$\frac{\mu(1+C)}{LD} + \log_2 D$	1			
PQ	$\frac{\mu(\log_2 N+C)}{LD} + \log_2 D$	$\log_2 N$	$\frac{1}{D} + \log_2 D$	C	$\frac{1}{L}$
CCT	$\frac{\mu(\log_2 N+C)}{LD} + \log_2 D$	$\log_2 N'$			

Table 4.3: Expected behaviour of manipulating model parameters

Thus the minimum expected events per LP execution is given by $E_{min} = LD/\mu$ regardless of the algorithm. Table 4.3 shows the per event cost expressed in terms of of N, L, C, D and μ. The second half of the Table shows the expected behaviour when the identified parameter is varied while keeping the others constant. These expected behaviours will be compared against the measured instruction counts obtained in Chapter 5. In Section 5.1.3, an experiment is conducted that maintains a constant event population of P events while varying the ratio of LPs to event density. The number of LPs parameter is manipulated while D is selected such that $N \times D = P$. The expected cost in terms of the number of LPs parameter is $N \log_2 N + N - \log_2 N$. This is easily obtained by substituting $\frac{P}{N}$ for D and taking all other parameters to be constant.

Recall that there are two types of event used in the synthetic workload model. The average lifetime of local events is determined by timestamp increment distribution which has a mean of 1. The average lifetime of external events is equal to the channel delta parameter L. Since each internal event is followed by an external event, the average lifetime of an event in the test model is given by $\mu = \frac{1.0+L}{2}$. In terms of model parameters, the minimum expected events per LP execution is $E_{min} = \frac{2LD}{1+L}$.

The maximum expected events per LP execution can also be computed. Relative to L, the channel lookahead an LP can advance at most $2L$ seconds of simulation time. If the simulation time of LP A is t, then the simulation times of its neighbours are at most $t + L$, since they would have blocked when they reached this time. This means

that LP A can safely advance up to the simulation time of its neighbors $t + L$, and then up to one channel delta ahead, i.e., to simulation time $t + 2L$. This implies that the maximum events per LP execution that can be achieved for any of the algorithms is $E_{max} = \frac{4LD}{1+L}$.

4.3.3 Comparison with CEL-based approaches

Parallel algorithms can conceivably achieve better cache performance under the assumption that event execution accesses the state of the LP. In terms of cache behaviour, a high event density does not guarantee better performance for the conservative algorithms; if the event density is high and localized, then both CEL and conservative algorithms can achieve good cache performance due to the localized behaviour. In this case both algorithms would exhibit events per LP execution greater than 1. However, when the event density is high and distributed, then the conservative algorithms are better able to achieve good cache performance, since they relax the total timestamp ordering imposed by CEL approaches. In this case only the conservative algorithms would achieve significantly greater than 1 event per LP execution.

There are many scenarios where event density is high and distributed; for example, packet based simulation of telecommunication networks. There are many events occurring across the system at any given time.

The behaviour of a CEL-based algorithm with logarithmic behaviour in event population is $O(\log_2 P = \log_2 ND = \log_2 N + \log_2 D)$. This is similar to the expression for channel based conservative algorithms, if the safetime calculation is ignored, i.e., $O(\frac{\log_2 N}{E} + \log_2 D)$. It is clear that for channel based conservative algorithms with events per LP execution greater than 1 the LP queue is sorted less frequently and the per event cost is reduced. This will also benefit the algorithm's cache behaviour. If the events per LP execution is less than 1, then the LP queue is sorted more frequently in comparison to a CEL-based algorithm with logarithmic behaviour in the event

population. An events per LP execution greater than 1 is a necessary condition for channel based conservative algorithms to achieve better asymptotic behaviour than CEL-based algorithms with $O(\log_2 P)$ behaviour.

4.4 Summary

This chapter has surveyed existing synthetic workload models and presented the test model that is used for experiments conducted in this thesis. The model was chosen for its ability to demonstrate the range of performance possible using channel based conservative techniques. The test model is implemented in the test system. Parameters of the test system are used to select the simulation kernel and simulation termination conditions. The test model parameters allow selection of three variations of the Chandy-Misra-Bryant algorithm, and five CEL implementations. The experimental methodology was discussed, including a description of performance metrics used for empirical comparison of the algorithms. The chapter also included theoretical analysis of the channel based conservative algorithms that will be compared with the simulation results in the following chapter.

Chapter 5

Sequential Performance of DES Algorithms

This chapter presents the results of the experiments conducted for this thesis. The experiments are grouped according to the parameters that they manipulate. Section 5.1 presents experiments that manipulate the event population, Section 5.2 presents experiments that manipulate the model topology including the connection radius and lookahead parameters. Section 5.3 presents experiments that manipulate computation grain, LP state size, and the timestamp increment distribution. Each section will review the parameters to be manipulated and identify the controlled variables. Table 5.1 summarizes the parameters used in the different experiments. An asterisk indicates the parameter that was manipulated for a particular experiment. The chapter concludes with a brief summary of the experiments conducted.

5.1 Queue Size Experiments

Varying the size of the event population is the standard approach that is used to evaluate the scalability of a simulation algorithm. This approach has been used in previous empirical studies of central event list implementations [27, 20], and in the

Experiment	Number of LPs	Event Density	Channel Delta	Connection Radius	Computation Grain	LP State Size	Topology	Distribution
Number of LPs	*	4	1.0	1	0.0	0	ring	uniform
Event Density	8192	*	1.0	1	0.0	0	ring	uniform
Fixed Queue Size	*	*	1.0	1	0.0	0	ring	uniform
Connection Radius	8192	4	1.0	*	0.0	0	ring	uniform
Lookahead	8192	4	*	1	0.0	0	ring	uniform
Topology	8192	4	1.0	*	0.0	0	*	uniform
	8192	4	*	4	0.0	0	*	uniform
Computation Grain	8192	4	1.0	4	*	0	ring	uniform
	16384	8	1.0	1	*	0	ring	uniform
LP State Size	8192	1	1.0	1	0.0	*	ring	uniform
	8192	4	1.0	1	0.0	*	ring	uniform
Timestamp Increment Distribution	8192	0.25	1.0	1	0.0	0	ring	*
	8192	32	1.0	1	0.0	0	ring	*
	8192	4	1.0	1	0.0	0	ring	*
	8192	4	1.0	32	0.0	0	ring	*
	8192	4	0.125	1	0.0	0	ring	*
	8192	4	2.0	1	0.0	0	ring	*

Table 5.1: Controlled variables for Queue size experiments

analysis of many parallel simulation algorithms [11, 23]. This group of experiments examines what happens to the performance of the various algorithms as the event population changes. In the test model, there are two parameters that affect the event population, the number of LPs (N) and the event density (D). The event density parameter determines the number of events initially generated for each LP, thus the total event population is equal to $N \times D$.

In the first experiment a constant event density of 4 is assumed and the number of LPs is varied between 16 and 65536. In the second experiment the number of LPs is fixed at 8192 and the event density varied from 1 to 64. Finally a third experiment assumes a constant event population of 131072 events and varies the ratio of the number of LPs to event density. See Table 5.1 for additional experiment parameters.

5.1.1 Number of LPs Experiment

This experiment assumes a constant event density of 4 while varying the number of LPs between 16 and 65536, resulting in queue sizes ranging from 64 to 262144 events. The purpose of this experiment is to study how the number of LPs affects the

performance of the different algorithms. The graphs in Figures 5.1 and 5.2 compare the observed metrics for the tested algorithms.

Figure 5.1A plots model level cache behaviour versus the number of LPs. For less than 128 LPs the entire model fits into the cache, resulting in nearly 0% cache misses. As the number of LPs increases from 128 to 1024, the percentage of cache misses climbs for both the conservative and CEL algorithms. For greater than 1024 LPs, the model level cache behaviour of the conservative algorithms is constant, while it continues to increase for the CEL algorithms. Estimating from the graphs, the model level cache behaviour of the CEL algorithms appears to be approaching a constant of about 5.5%. The CCT algorithm remains at about 1%, the fixed schedule algorithm at 1.5%. The CCT and CMB algorithms schedule LPs based on local virtual time or critical channels in order to avoid scheduling LPs which are blocked. Since the fixed schedule algorithm pays no attention to whether an LP is a good candidate for execution, it is more likely to schedule an LP that has no events safe to execute. This negatively affects cache performance by performing unnecessary loads into the cache. In the context of this experiment, it is clear that the conservative algorithms are superior to the CEL algorithms in terms of model level cache behaviour.

As can be observed in the graph for kernel level cache behaviour (Figure 5.1B), the conservative algorithms achieve a constant cache miss percentage in the number of LPs. This is approximately 2.5% for the fixed schedule algorithm and 1.7% for the CCT algorithm. The calendar queue algorithm demonstrates the next best kernel level cache behaviour which appears sub-logarithmic in the number of LPs. The heap and splay tree algorithm are approximately logarithmic, and Henriksen's algorithm increases faster than logarithmically over the observed range. For 65536 LPs, the CCT algorithm has less than one half the cache miss percentage of the calendar queue, and one seventh that of Henriksen's algorithm. The conservative algorithms demonstrate superior performance over CEL algorithms in terms of kernel level cache behaviour. Although kernel level cache behaviour of the conservative algorithms

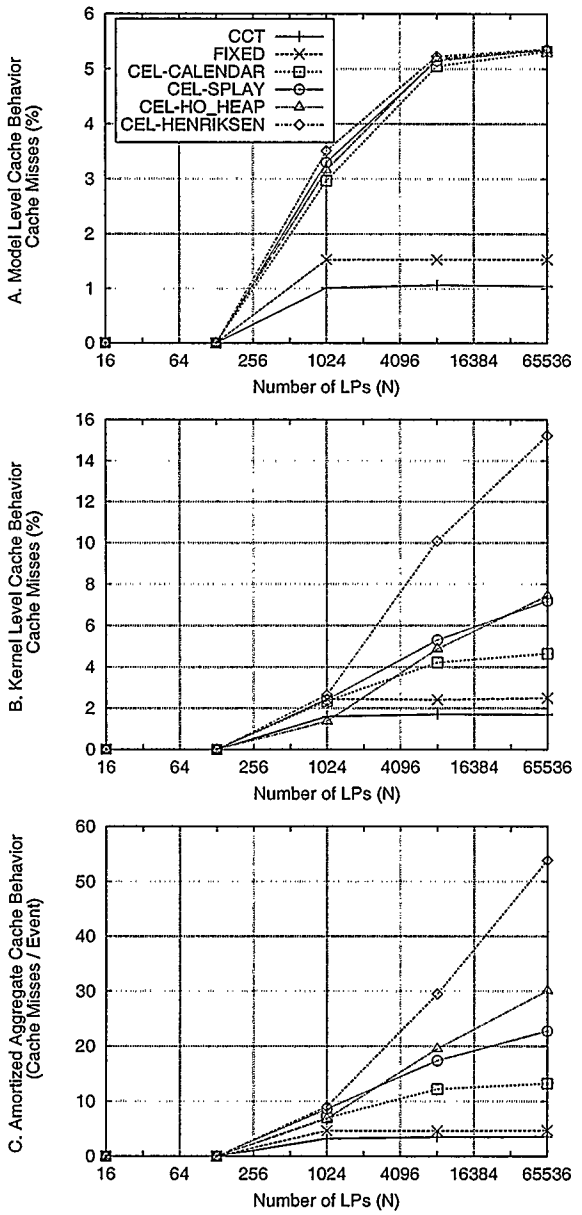


Figure 5.1: Number of LPs experiment.Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus the number of LPs.

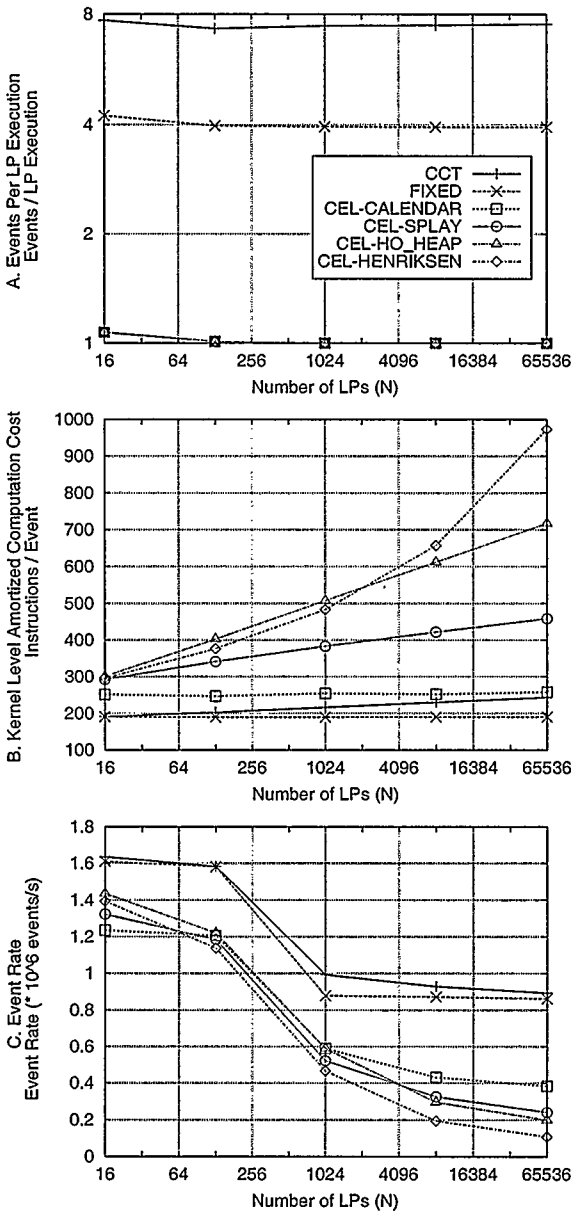


Figure 5.2: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus the number of LPs.

appears constant in the number of LPs, this is probably due to characteristics of the model and implementation of the LP scheduling queue. For a different synthetic workload model, a slightly increasing cache miss percentage might be observed.

Figure 5.1C shows the combined affect of model and kernel level cache behaviour in terms of the number of cache misses per event executed. For 65536 LPs, there are over 3 times more cache misses involved with processing an event using the calendar queue algorithm, than when using the CCT algorithm and over 15 times as many when using Henriksen's algorithm. The superior cache performance of the conservative algorithms is due to a large extent to the greater number of events executed per LP execution session.

The number of events per LP execution is plotted in Figure 5.2A. Analysis of the conservative algorithms in section 4.3 suggested that the number of events per LP execution was independent of the number of LPs in the model, this is confirmed by the plot. Recall the formulas for the E_{min} and E_{max} derived for the synthetic workload model in section 4.3. In this experiment the fixed schedule algorithm is observed achieving the minimum expected events per LP execution of about $E_{min} = \frac{DL}{\mu} = \frac{DL}{\frac{1+L}{2}} = \frac{2\times4\times1}{1+1} = 4$. CCT achieves nearly the optimal expected behaviour of $E_{max} = \frac{2DL}{\mu} = \frac{2DL}{\frac{1+L}{2}} = \frac{2\times2\times4\times1}{1+1} = 8$.

Figure 5.2B plots the average number of kernel level instructions required to process a simulation event. The asymptotic behaviour of the algorithms breaks down into three categories: radical, logarithmic and constant. The kernel level computation cost grows faster than logarithmically for Henriksen's algorithm, this follows the $O(\sqrt{N})$ expected behaviour. The behaviour of the calendar queue and fixed schedule algorithms remain constant in the number of LPs, while the heap, splay tree and CCT algorithms exhibit logarithmic behaviour. As predicted by the theoretical analysis in section 4.3, the behaviour of the CCT algorithm is logarithmic in the number of LPs. This dependence does not occur for the fixed schedule algorithm since the scheduling queue operations are $O(1)$, rather than $O(\log N')$. The fixed schedule conservative

algorithm achieves a kernel level computation cost of approximately 190 instructions per event, the calendar queue is 37% more expensive at 260 instructions per event. Although the cost of the CCT algorithm is increasing, it remains less than that of the calendar queue for up to 65536 LPs. For 65536 LPs, the cost of heap, splay and Henriksen's algorithms are significantly greater and increasing faster than for the calendar queue or conservative approaches.

As can be observed in Figure 5.2C, the event rate of all algorithms decreases rapidly as the number of LPs grows from 128 to 1024. This is a result of the model becoming larger than the size of the cache. For models smaller than 128 LPs, essentially no cache misses were observed. For greater than 1024 LPs, the fixed schedule algorithm achieves a nearly constant event rate of about 860,000 events per second. As noted in Figure 5.2B, the per event cost of the CCT algorithm is logarithmically increasing in the number of LPs. This explains why the event rate of the CCT algorithm continues to decline. The event rate of the CEL algorithms also declines, but more rapidly than for the conservative algorithms. For 65536 LPs, the conservative algorithms are observed to be over 2 times faster than the calendar queue algorithm. The variation of performance between the CEL algorithms is quite large, Henriksen's algorithm achieves about 108,000 events per second, while the calendar queue achieves around 385,000.

5.1.2 Event Density Experiment

In this set of experiments event density is manipulated in order to study its effect on the performance of the different algorithms. The number of LPs is fixed at 8192 and the event density is varied between 0.25 and 64. This combination of parameters will result in queue sizes ranging from 2048 to 524288. Like increasing the number of LPs, increasing the event density has the effect of increasing the size of the event list and the memory footprint of the model. Figures 5.3 and 5.4 compare the different algorithms.

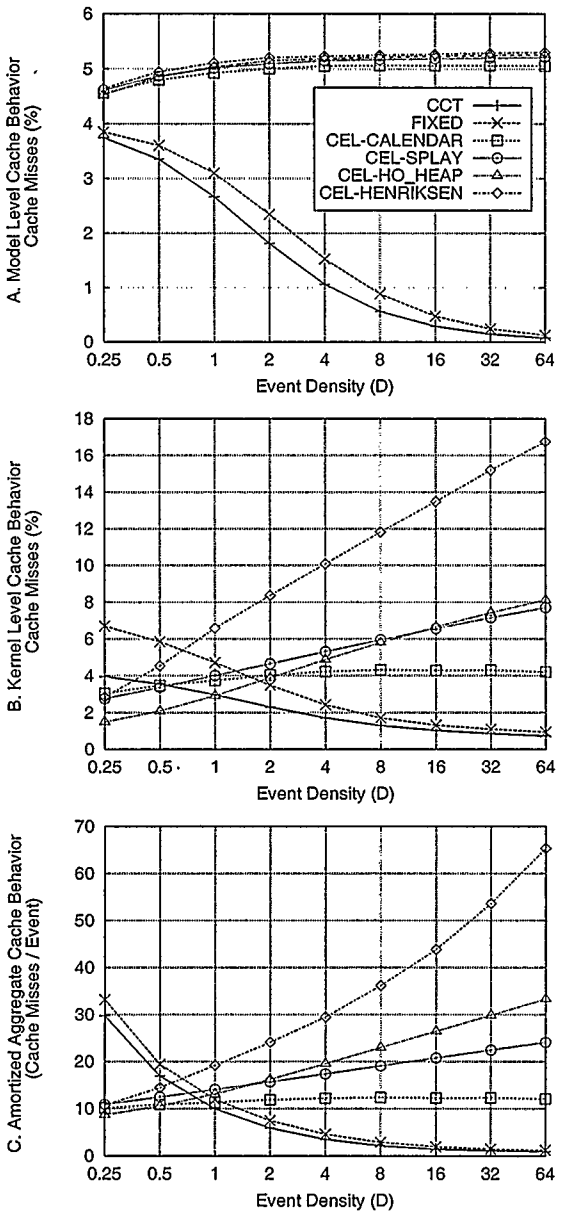


Figure 5.3: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus event density.

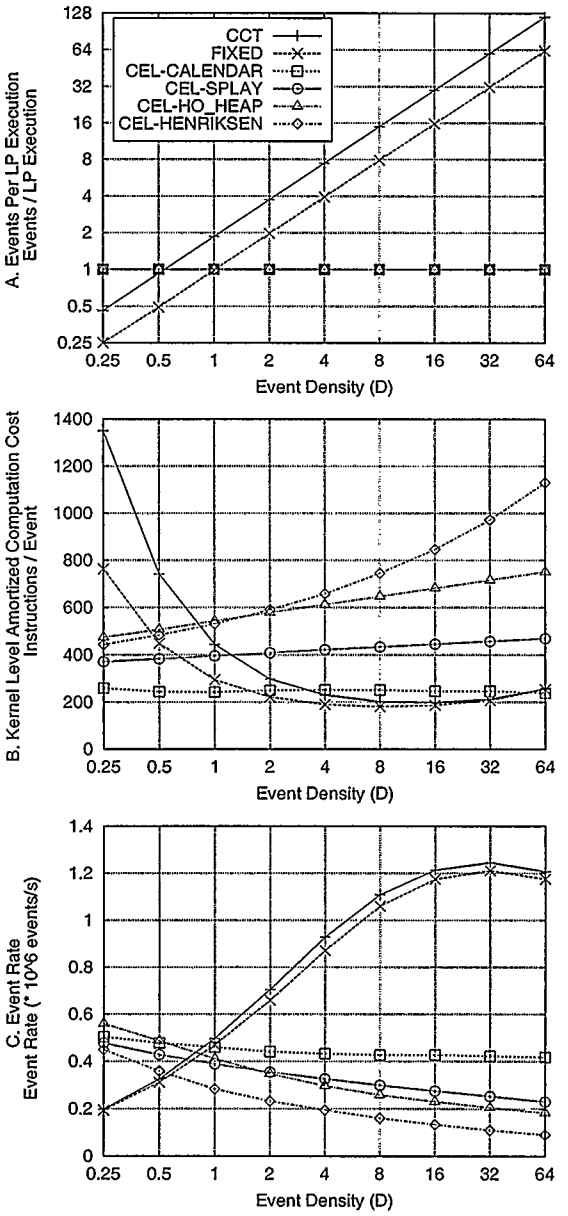


Figure 5.4: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus event density.

Figure 5.3A plots model level cache behaviour versus the event density. The model level cache behaviour of the CEL algorithms is independent of the priority queue implementation, i.e., the cache behaviour is the same for all CEL algorithms. For the conservative algorithms there appears to be some dependence on the LP scheduling method, this can be seen from the plot of the CCT algorithm which performs slightly better than the fixed schedule algorithm for moderate event densities. For very large or very small event densities there seems to be less dependence on the LP scheduling mechanism. For small event densities the variation in cache behaviour is less evident due to the presence of a low-lookahead cycle. For large event densities the overhead of using a linked list to implement the local event queue hides the variation between the fixed schedule and CCT algorithms. The model level cache behaviour of the CEL algorithms is relatively independent of the event density, and is just over 5% for event densities greater than 2. The model level cache behaviour of the conservative algorithms improves rapidly with increasing event density. For an event density of 0.25 the percentage of cache misses is near 4%, for event densities greater than 8 it is consistently less than 1% (approaching 0 % cache misses). Even for small event densities (less than 1), the conservative algorithms maintain better model level cache behaviour than the CEL algorithms.

As can be observed in Figure 5.3B, the kernel level cache behaviour of most of the CEL algorithms deteriorates logarithmically with increasing event density. The calendar queue implementation has nearly constant kernel level cache behaviour of about 4% cache misses. The kernel level cache behaviour of conservative algorithms improves as event density increases and is less than 4% for event densities greater than 2. For an event density of 64, the percentage of cache misses for the conservative algorithms is less than 1/5 that of the calendar queue algorithm, and less than 1/20 that of Henriksen's algorithm. The kernel level cache behaviour of the conservative algorithms can be poor if the event density is less than 1. This is due to the presence of low-lookahead cycles which prevent the execution of the next event until LP clocks

have advanced.

The overall cache performance of the algorithms is compared in Figure 5.3C. The number of cache misses per event is approximately constant for the calendar queue algorithm, increasing logarithmically for the heap and splay tree algorithms, and increasing faster than logarithmically for Henriksen's algorithm. For the conservative algorithms, the number of cache misses decreases exponentially in the event density. For an event density of 0.25, the conservative algorithms suffer about 30 cache misses in processing a single event, for event densities greater than 1 this drops below 10 cache misses per event. For an event density of 64, there is approximately 1 cache miss associated with processing an event using a conservative approach, 12 misses using the calendar queue algorithm and 65 misses using Henriksen's algorithm.

Figure 5.4A plots the events per LP execution for the tested simulation algorithms. As previously discussed the metric is very close to 1 for all CEL algorithms. The fixed schedule algorithm achieves the minimum expected events per LP execution $E_{min} = D$, while the CCT algorithms achieve nearly twice that value at $E_{max} = 2D$. The improvement in cache behaviour is explained by the increasing events per task LP execution observed with increasing event density.

The performance of the algorithms in terms of kernel level amortized computation cost is plotted against event density in Figure 5.4B. The number of kernel level instructions required to process an event is approximately constant for the calendar queue algorithm, independent of the event density. The heap and splay tree algorithms exhibit a logarithmic dependence on the event density, with the number of instructions growing slightly slower for the splay tree than for the heap. Henriksen's algorithm performs poorly, the computation cost is in $O(\sqrt{D})$. The behaviour of the conservative algorithms matches the $O(D + \frac{1}{D})$, predicted in section 4.3. For event densities less than 8, the $O(1/D)$ term dominates the behaviour. For event densities greater than 8, the behaviour is affected more by the $O(D)$ term. As discussed previously, this second term depends on the implementation of the local event queue. If a

heap structure was used the cost would increase in $O(\log D)$, rather than $O(D)$. For event densities less than 8, the kernel level amortized computation cost of the fixed schedule algorithm can be significantly less than for the CCT algorithm even though it fails to achieve the same number of events per LP execution. For event density 64, Henriksen's algorithm requires about 5 times the number of instructions as does either conservative approach, the calendar queue requires about 2 times as many.

Figure 5.4C plots event rate versus event density for each algorithm. The event rate of the CEL algorithms decreases as the event density increases, the rate of decrease is greatest for Henriksen's algorithm, followed by the heap, splay and calendar queue algorithms. The event rate of the conservative algorithms actually improves with increasing event density until the cost of managing the local event queue becomes too large, i.e., when the $O(D)$ term begins to dominate the kernel level computation cost. Actually, although the computation cost begins to increase at event density 8, the effect of the increasing cost is not observed in the event rate metric until event density 32, this is due to the cache behaviour of the conservative algorithms which continues to improve beyond event density 8, offsetting the increased computation cost. For event densities of 1 or greater the conservative algorithms are faster than the CEL algorithms. For event density 32, the conservative algorithms achieve over 1.2 million events per second and are about 3 times faster than the calendar queue algorithm and about 6 times faster than Henriksen's algorithm.

5.1.3 Fixed Queue Size Experiment

The purpose of this experiment is to examine the effect of the ratio of LPs to event density. The queue size is fixed at 131072 events, while the number of LPs and the event density are varied such that the queue size remains constant. The number of LPs is increased from 2 to 65536, while the event density is chosen such that $N \times D = 131072$. The conservative algorithms in this experiment were tested using a hold optimized heap for the local event queue implementation. The cost of using a

linked list for the local event queue would be excessive for the queue sizes encountered in this experiment.

The graphs in Figures 5.5 and 5.6 show the entire spectrum of performance for each algorithm and partitioning of the model into LPs. The smallest model contains 2 LPs each with about 65536 events, the largest model contains 65536 LPs with an average of 2 events per LP. The extreme cases with 131072 LPs and 1 event per LP, and 1 LP with 131072 events were not tested since they are equivalent to CEL simulation. The performance in the case of a single LP would be similar to the hold optimized heap since there would be very little additional synchronization overhead. The case of 131072 LPs would have asymptotic behaviour similar to the hold optimized heap but would incur additional overhead due to the synchronization protocol.

Figure 5.5A plots model level cache miss behaviour versus the number of LPs, with the event density parameter selected to obtain a population of 131072 events. Although the number of events is constant, increasing the number of LPs has the effect of increasing the memory footprint of the model. For up to 16 LPs, most of the simulation fits into memory and thus relatively few cache misses are observed. As the number of LPs increases from 16 to 16384, the percentage of cache misses increases rapidly for the CEL algorithms. The onset of increasing model level cache misses is delayed for the conservative algorithms due to the large number of events per LP execution (see Figure 5.6A). Even for 65536 LPs, the model level cache behaviour is still 2 times better than that of any CEL algorithm.

The increasing memory footprint of the model is also observed in the graph of kernel level cache behaviour (Figure 5.5B), particularly for the CEL algorithms which are otherwise unaffected by the number of LPs. The conservative algorithms show a decrease in the percentage of cache misses as the number of LPs increases from 2 to 16. At this extreme, i.e, when the number of LPs equals 2, the conservative algorithms have degenerated to almost a CEL approach. The cache behaviour of the simulator now depends on the implementation of the LP's local event list, in this case,

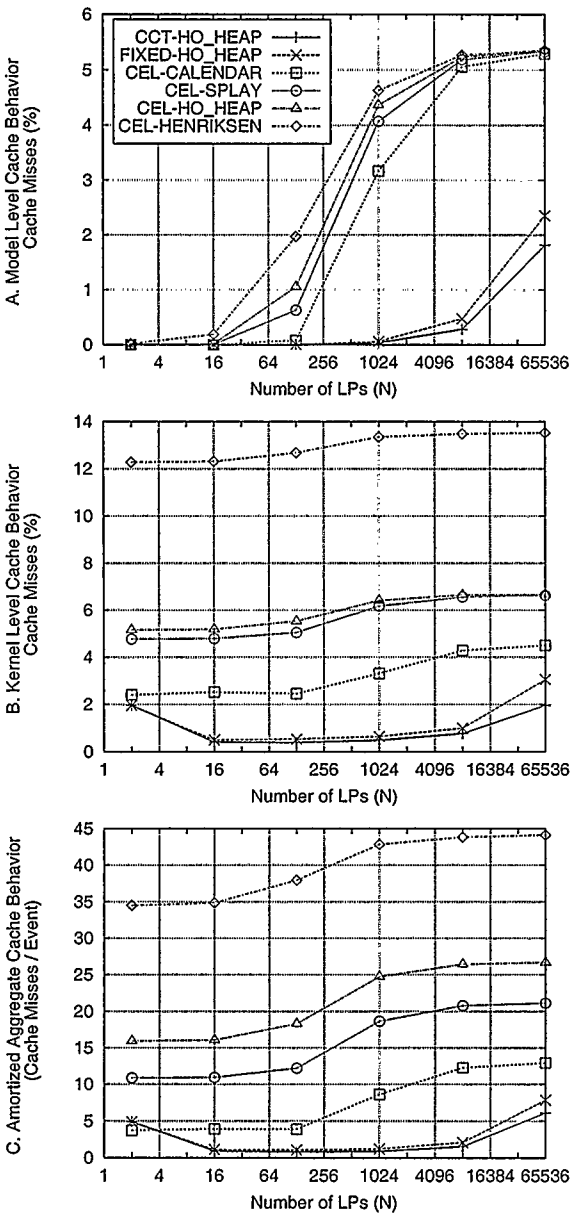


Figure 5.5: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus the number of LPs. Parameter D chosen such that $N \times D = 131072$.

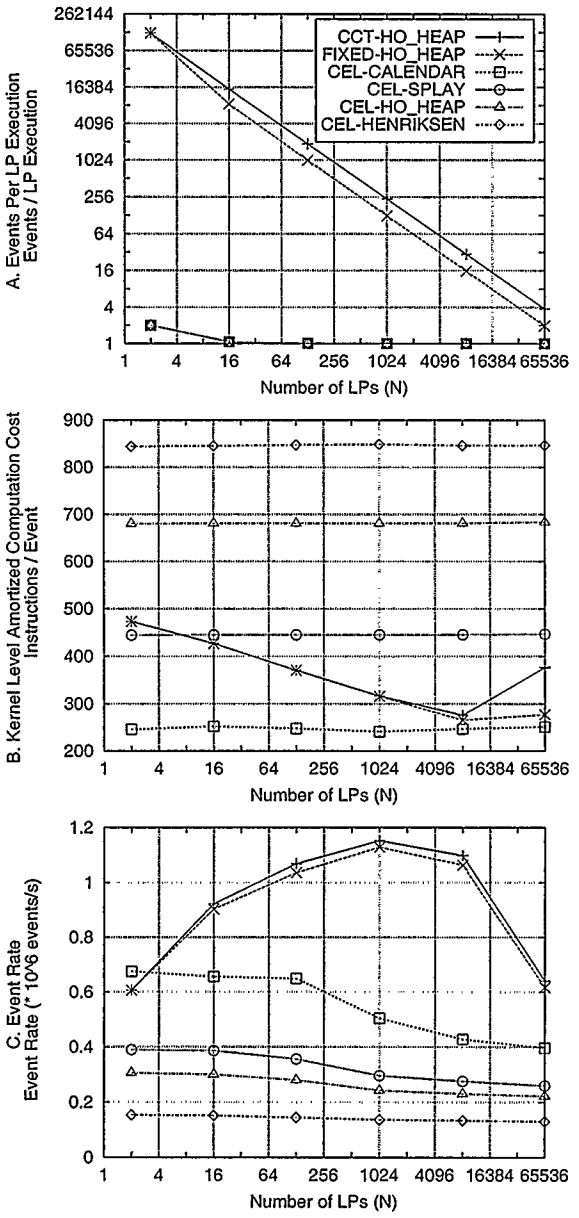


Figure 5.6: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus the number of LPs. Parameter D chosen such that $N \times D = 131072$.

2 hold optimized heaps of 65536 events each. When the size of the model increases an increase in the percentage of kernel level cache misses for conservative algorithms is also observed. This is due to the increasing memory footprint and the decreasing number of events per LP execution (Figure 5.6A). The kernel level cache behaviour of the conservative algorithms remains better than for CEL algorithms throughout.

Discussion of the aggregate cache miss behaviour (Figure 5.5C) follows that for the kernel level cache behaviour above. The calendar queue achieves better overall cache performance when the ratio of the number of LPs to events is very small (less than 4:32768). Although not entirely obvious from the plot, the cache misses per event increase by 67% from 0.9 to 1.5 as the number of LPs increases from 1024 to 8192. For 1024 LPs, the number of cache misses per event using the CCT algorithm is about 1/10 that when using the calendar queue and about 1/50 that when using Henriksen's algorithm.

The events per LP execution plots are shown in Figure 5.6A. For 2 LPs, the CEL algorithms achieve 2 events per LP execution since each LP executes a single event with 100% probability, then a second event with 50% probability, then a third event with 25% probability, etc. A rapid decline in events per LP execution is observed for the conservative algorithms, this due to the decreasing event density. For this experiment $E_{min} = \frac{DL}{\mu} = \frac{\frac{QL}{N}}{\frac{1+L}{2}} = \frac{2\times131072\times1}{N(1+1)} = \frac{131072}{N}$ and $E_{max} = \frac{2DL}{\mu} = \frac{\frac{2QL}{N}}{\frac{1+L}{2}} = \frac{2\times2\times131072\times1}{N(1+1)} = \frac{262144}{N}$. The CCT algorithm achieves nearly optimal behaviour, while the fixed schedule algorithm achieves about half that. The fixed schedule does obtain the expected maximum events per LP execution when the number of LPs is 2. This occurs because the only permutation of 2 LPs is optimal in the sense that each LP will always advance one lookahead interval up to the time of its neighbor and then one lookahead interval beyond.

It is evident in Figure 5.6B that the kernel level computation cost of CEL algorithms is independent of the ratio of the number of LPs to event density. This makes sense since the cost of these algorithms is only dependent on the number of events in

the central event list which remains constant. The asymptotic behaviour of the conservative algorithms is more complicated. Assuming a heap is used for both the LP scheduling mechanism and the local event queue implementation, the computation cost is in $O(N \log_2 N + N - \log_2 N)$ (Section 4.3). As evidenced by the decreasing kernel level computation cost, the $-\log N$ dominates for up to 8192 LPs. The kernel level computation cost for conservative algorithms begins to increase at 8192 LPs. The increase is greater for the CCT algorithm since it must maintain an increasing number of LPs in the scheduling queue.

Figure 5.6C plots event rate versus the ratio of the number of LPs to event density. The event rates for CEL algorithms are governed by their respective cache performance. As the memory footprint of the model increases, the event rate decreases in proportion to the number of cache misses per event. The event rates of the conservative algorithms increase from about 2 to 1024 LPs after which they begin to decrease. From the kernel level computation cost one would expect the optimal ratio of LPs to event density to be higher, i.e., 8192 LPs. The reason the optimal performance is observed for 1024 LPs is likely due to the better cache performance at the smaller model size. There are 67% fewer cache misses per event at 1024 LPs and 15% more instructions as compared to 8192 LPs. For 8192 LPs, the event rate of the CCT algorithm is approximately 2.6× that of the calendar queue algorithm.

5.2 Model Topology Experiments

In this group of experiments, the topological characteristics of the model will be varied. Three parameters have been introduced to allow simulation of a wide variety of topologies. The behaviour of the CEL algorithms was not significantly affected by the model topology parameters. Results are presented comparing the conservative algorithms to the calendar queue algorithm. The first two experiments include simulation runs with event density 1 and event density 4. This shows how event density

affects the sensitivity of the conservative algorithms to the channel delta and connection radius parameters. Table 5.1 contains the controlled variables used in the Model Topology experiments.

The first parameter is the channel delta, this will determine the available amount of lookahead between neighboring LPs. The significance of lookahead to the performance of PDES algorithms has been extensively researched, and was discussed in Chapter 3. Limiting the available lookahead when event density is small, is likely to have a significant impact on the conservative algorithms. In this experiment the channel delta is varied between 0.015625 and 4, while keeping the connection radius fixed at 1.

The second topology related parameter is the connection radius, this parameter controls the number of neighbors that each LP is connected to. Increasing the connection radius is expected to have a negative impact on the performance of the conservative algorithms. The reason for this is that the safetime calculation performed at the beginning of each LP execution session is dependent on the number of channels. The connection radius may impact the model in other ways since it will affect the movement of events throughout the entire system. This experiment will consider the effect of varying the connection radius with the channel delta fixed at 1.0 unit of simulation time.

The third parameter manipulated in this section selects either the ring or star topology as described in section 4.1.7. Each algorithm is tested for both topologies. Two experiments are conducted. In the first experiment channel delta is manipulated, in the second experiment connection radius is manipulated. The event density is kept constant at 4 for both experiments.

5.2.1 Channel Delta Experiment

The purpose of this experiment is to observe the effect of channel delta on the performance of the different algorithms. The channel delta should have little impact on

the performance of CEL algorithms, although it will affect the model behaviour and could affect performance that way. Reducing the channel delta is expected to have a negative impact on the conservative algorithms, since this will reduce the number of events per LP execution. For this set of tests the connection radius is fixed at 1, and the channel delta is varied between 0.015625 and 4.0. Performance metrics are graphed in Figure 5.7 and 5.8.

In terms of model level cache behaviour (Figure 5.7A), the CEL algorithms appear insensitive to the channel delta. The slight downward trend observed with increasing channel delta is due to the model not reaching a steady state by the specified simulation end time. The channel delta affects the timestamp increment function, and thus how many events are executed prior to the simulation end time. The downward trend is not observed if the termination condition is changed to require the execution of a certain number of events. Increasing channel delta improves the model level cache behaviour of the conservative algorithms. When event density is 1, the CCT algorithm exhibits better model level cache behaviour for channel delta greater than 0.25. When event density is 4, the CCT algorithm exhibits better model level cache behaviour for channel delta greater than 0.0625. As channel delta decreases, the model level cache behaviour of the conservative algorithms declines to approximately the level of the calendar queue algorithm.

The kernel level cache behaviour plotted in Figure 5.7B is approximately constant for CEL algorithms, suggesting that their cache performance is unaffected by channel delta. The performance of the conservative algorithms is dependent on channel delta, with kernel level cache performance improving with increasing channel delta. This improvement is due to the increased number of events per LP execution (see Figure 5.8A). For event density 1 the kernel level cache miss percentage for the CCT algorithm is better than for the calendar queue algorithm, provided the channel delta is greater than 0.125. For a given event density the kernel level cache behaviour of the CCT algorithm is consistently better than that of the fixed schedule algorithm.

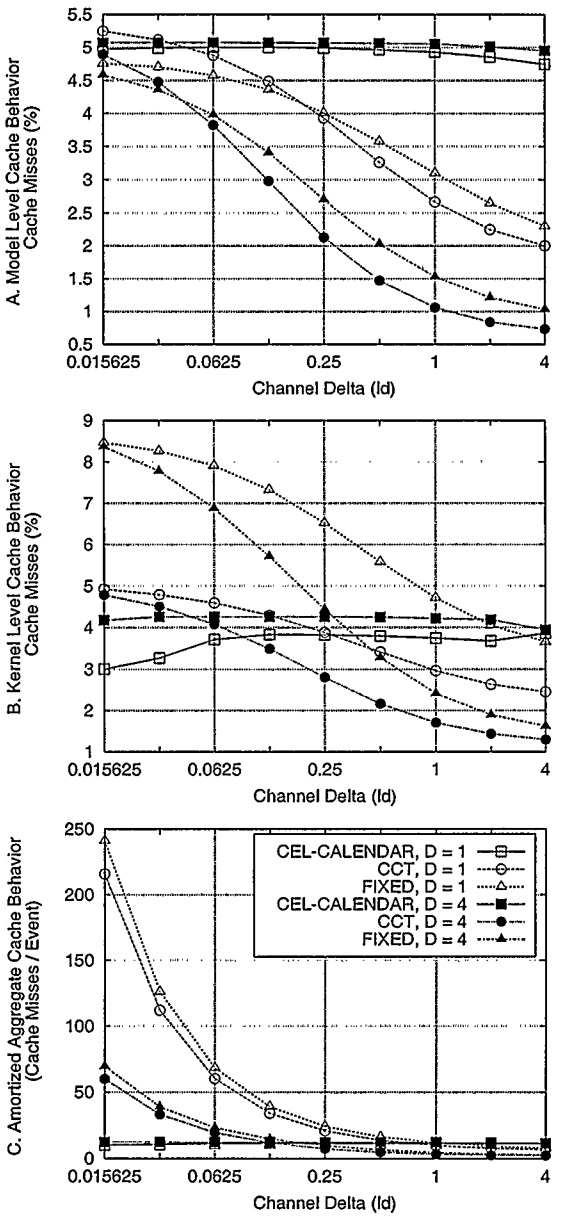


Figure 5.7: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus channel delta.

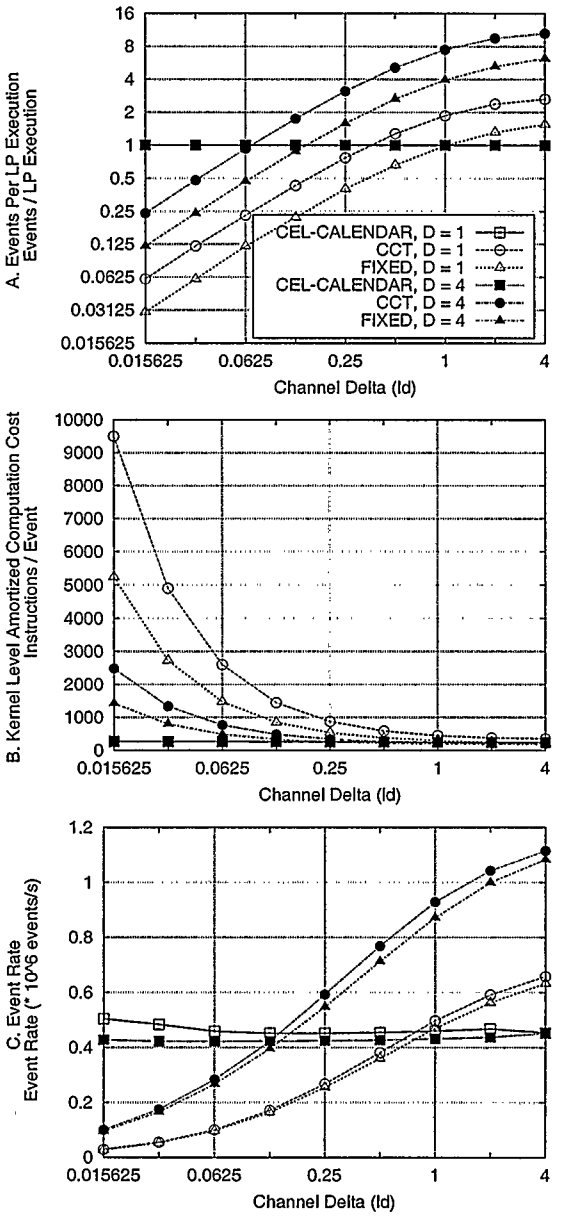


Figure 5.8: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus channel delta.

Figure 5.7C gives a full picture of the per event cache behaviour of the different algorithms. Again independence of CEL algorithms with regard to channel delta is observed. The cost of using a conservative algorithm is quite severe when channel delta is small; however, this expense can be moderated by a larger event density as is observed in the graph. In a model were the event density is 1 and the channel delta is 0.015625, the fixed schedule algorithm experiences 241 cache misses during the execution of a single event. Increasing the event density to 4 reduces the number of cache misses to about 70 per event. For the conservative algorithms, the number of cache misses per event rapidly decreases with increasing channel delta and eventually becomes less than that of the calendar queue, how quickly this happens depends on the event density of the model.

Figure 5.8A plots the number of events per LP execution versus the channel delta. Again, this metric is consistently 1 for the CEL algorithms. For the conservative algorithms, the fixed schedule algorithm achieves the expected minimum events per LP execution. $E_{min}(D=1) = \frac{DL}{\mu} = \frac{1\times L}{\frac{1+L}{2}} = 2L/(1+L)$, $E_{min}(D=4) = \frac{DL}{\mu} = \frac{4\times L}{\frac{1+L}{2}} = 8L/(1+L)$. For both D=1 and D=4, the CCT algorithm achieves nearly the expected maximum events per LP execution. $E_{max}(D=1) = \frac{2DL}{\mu} = \frac{2\times 1\times L}{\frac{1+L}{2}} = 4L/(1+L)$, $E_{max}(D=4) = \frac{2DL}{\mu} = \frac{2\times 4\times L}{\frac{1+L}{2}} = 16L/(1+L)$.

Figure 5.8B plots kernel level computation cost against channel delta. This graph is very similar to the graph of amortized aggregate cache behaviour in Figure 5.7C. Again, the CEL algorithms are unaffected by channel delta, whereas the conservative algorithms are extremely sensitive to channel delta, this is particularly obvious for channel deltas less than 1. The difference that can be seen here is in the relative performance of the CCT and fixed schedule algorithms. In terms of cache behaviour CCT had better performance than the fixed schedule algorithm, whereas in terms of kernel level computation the fixed schedule requires fewer instructions per event than the CCT algorithm. The behaviour of the conservative algorithms is 1/L, where L is the lookahead. The horizontal and vertical asymptotes, $y = 0$ and $x = 0$ can be easily

observed in the graph. In other words, the cost of the simulation increases rapidly as the lookahead approaches 0, and vanishes as the lookahead approaches infinity. If the lookahead is as long as the simulation itself, then in some sense there is no simulation to execute, and thus the cost is zero.

The event rate graph (Figure 5.8C) plots the number of events executed per second of wall-clock versus the channel delta. The performance of the calendar queue algorithm is approximately constant in the channel delta. The conservative algorithms do well when the channel delta is large, and poorly when the channel delta is small. For the range of channel deltas tested in this experiment when event density is low (D=1), the conservative algorithms can be up to 17× slower than the calendar queue algorithm, and up to 2× faster. When event density is larger (D=4), the conservative algorithms can be about 4× slower than the calendar queue algorithm and up to 2.5× faster. The point at which the conservative algorithms gain advantage of the CEL algorithms depends both on the lookahead and the event density. A necessary condition evident from the graphs, is that the events per LP execution greater for conservative algorithms must be greater than 1 to achieve better performance than the calendar queue algorithm, i.e., when $DL/\mu > 1$, it is possible for the conservative approach to outperform the optimal CEL approach. It is interesting that the CCT algorithm achieves better cache performance than the fixed schedule algorithm, due in part to maximizing the events per LP execution. However, the kernel level computation cost is still lower for the fixed schedule. In the end the modest cache advantage is enough to put the CCT algorithm ahead of the fixed schedule in terms of event rate.

5.2.2 Connection Radius Experiment

The purpose of this experiment is to study the effect of connection radius on the performance of the different algorithms. Fixing the channel delta at 1.0 units of simulation time, the connection radius is varied between 1 and 32. The experiment is conducted with event densities 1 and 4 to help illustrate how greater event densities

can offset the increased costs of a larger connection radius.

Figure 5.9A plots the percentage of level 2 cache misses in model level code versus the connection radius. The model level cache behaviour of the CEL algorithms appears unaffected by connection radius. With increasing connection radius the percentage of cache misses for conservative algorithms is observed to increase, although slower than logarithmically. The model level cache behaviour of conservative algorithms is consistently better than that of CEL algorithms for connection radius between 1 and 32, even when D=1. For the conservative algorithms, a higher event density results in significantly lower cache misses as can be seen from curves where D=4. The fixed schedule algorithm outperforms the CCT algorithm for a connection radius of 4 or greater. This is probably due to the CCT algorithm achieving more events per LP execution than the fixed schedule algorithm for smaller connection radius (See Figure 5.10A). For connection radius of 4 or more there is little difference between the events per LP execution of the fixed schedule and CCT algorithm.

The kernel level cache behaviour plotted in Figure 5.9B shows the independence of CEL algorithms with respect to connection radius. The conservative algorithms must perform a safetime calculations that requires accessing a different memory location for each channel. If none of these channels are cached then executing the first event will require $2R$ cache misses.

The percentage of kernel level cache misses increases faster than logarithmically for the CCT algorithm. It increases approximately logarithmically for the fixed schedule algorithm when D=4, and slower than logarithmically when D=1. The fixed schedule algorithm outperforms the CCT algorithm for connection radii larger than 4. When event density equals 1 the percentage of kernel level cache misses for conservative algorithms is almost immediately larger than for CEL algorithms, for event density 4 the conservative algorithms maintain a cache advantage up to a connection radius of 8.

The amortized aggregate cache behaviour can be observed in Figure 5.9C. The

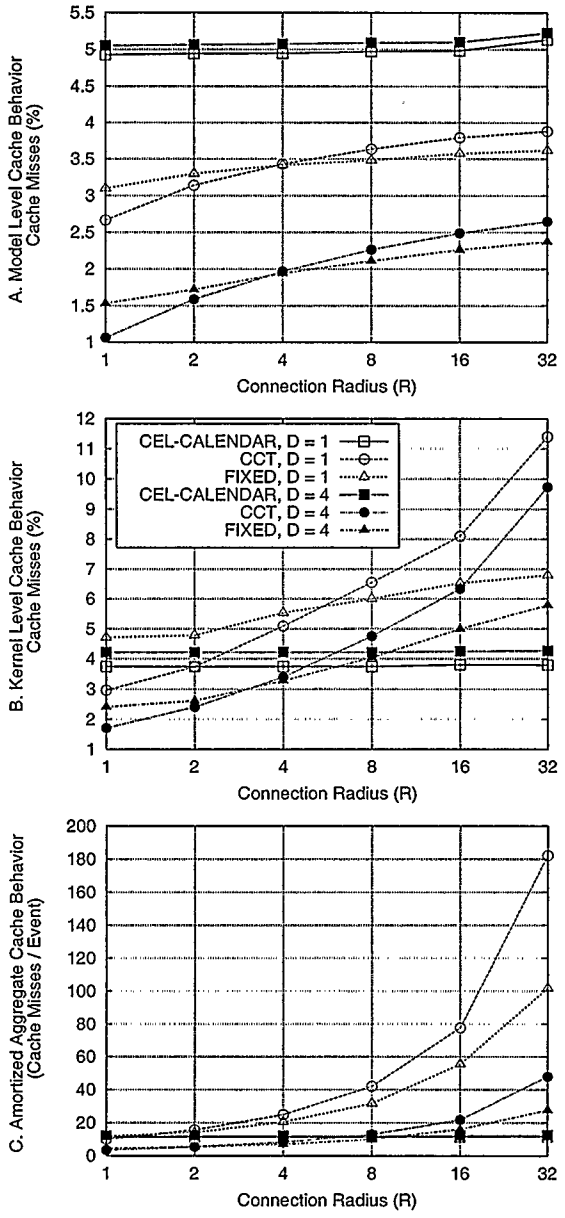


Figure 5.9: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus connection radius.

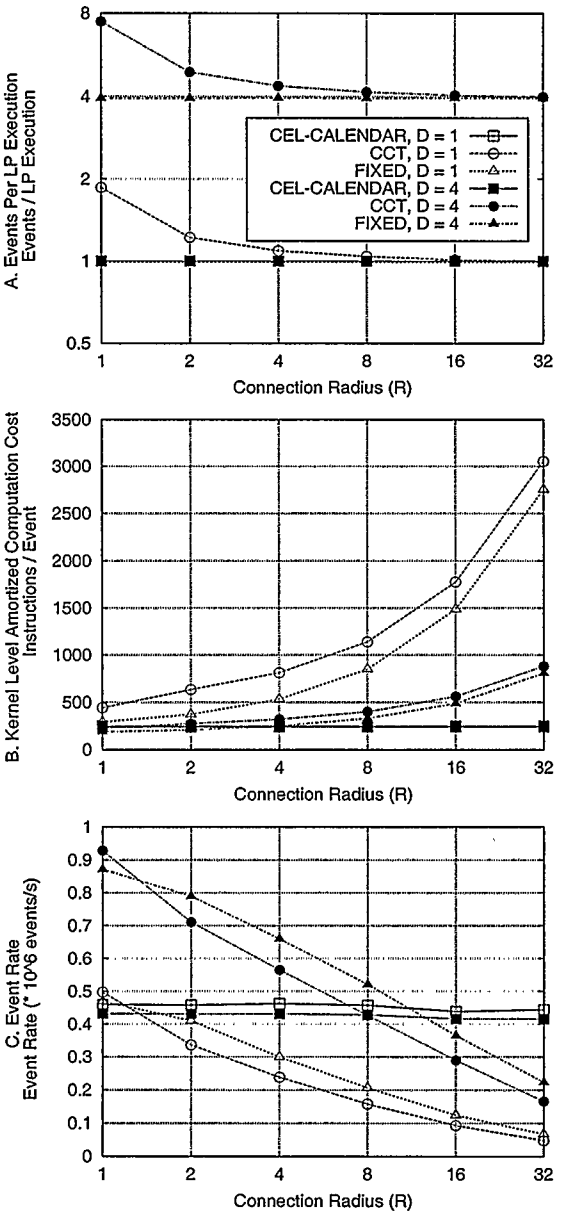


Figure 5.10: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus connection radius.

behaviour of the calendar queue algorithm is constant in the number of LPs. The conservative algorithms show significant increases in the number of cache misses per event as the number of channels in the model increases. These increases are greatly reduced with larger event densities, as can be seen comparing the curves for D=1 and D=4. The rate of increase is approximately linear in the connection radius. The fixed schedule algorithm consistently experiences fewer cache misses per event than the CCT algorithm. For event density 4, the conservative algorithms achieve better cache performance than the calendar queue algorithm for a connection radius up to 8. The calendar queue demonstrates better amortized aggregate cache behaviour for any connection radius larger than 8.

Figure 5.10A plots the number of events per LP execution. This metric is consistently 1 for the CEL algorithms. For the conservative algorithms, the fixed schedule algorithm achieves the minimum expected events per LP execution. $E_{min}(D = 1) = \frac{DL}{\mu} = \frac{1\times1}{\frac{1+1}{2}} = 1$, $E_{min}(D = 4) = \frac{DL}{\mu} = \frac{4\times1}{\frac{1+1}{2}} = 4$. The CCT algorithm is capable of achieving greater events per LP execution, but for larger connection radius this advantage quickly disappears. $E_{max}(D = 1) = \frac{2DL}{\mu} = \frac{2\times1\times1}{\frac{1+1}{2}} = 2$, $E_{max}(D = 4) = \frac{2DL}{\mu} = \frac{2\times4\times1}{\frac{1+1}{2}} = 8$. With a larger connection radius, there is a greater chance that one of the neighbors may not have advanced ahead of the currently executing LP (i.e., the currently executing LP and one of its neighbors have the same timestamp at the beginning of the execution session), this means that the executing LP will only advance a single lookahead interval.

Kernel level computation cost is plotted in Figure 5.10B. Again, the behaviour of CEL algorithms is observed to be constant in the connection radius. The number of kernel level instructions per event increases linearly for the conservative algorithms. This confirms the analysis of section 4.3. The rate of increase of this metric is about the same regardless of event density, but the value of this metric is offset depending on the event density parameter. For a connection radius of 4 or greater, the kernel level computation cost is greater for conservative algorithms than for the calendar

queue algorithm. The conservative algorithms must perform a safetime calculation, which is theoretically a function of the number of channels, this cost is not incurred by CEL algorithms which explains the difference in behaviour observed here.

The event rate metric is plotted in Figure 5.10C. The behaviour of CEL algorithms is almost constant, although there is a very slight downward trend possibly due to other interactions in the model. When the event density is equal to 1 the performance of the conservative algorithms is poor in comparison to the calendar queue algorithm. The conservative algorithms achieve comparable performance for connection radius 1, and then worse performance as the connection radius is increased. With a larger event density such as D=4, the conservative algorithms are able to outperform the calendar queue algorithm for a greater range of connection radius. However, their performance is still declining relative to the CEL algorithms. For a connection radius of 2 or greater, the fixed schedule conservative algorithm consistently performs better than the CCT algorithm. One would expect CCT to do better for larger connection radius by maximizing the events per LP execution. The reason this has not been observed is that the fixed schedule always achieves E_{min} events per LP execution, while the CCT algorithm only achieves greater than E_{min} for small connection radius. Although the behaviour of the fixed schedule and CCT algorithms is similar in terms of events per LP execution, the kernel level computation cost is lower for the fixed schedule and thus its event rate is greater.

5.2.3 Topology Experiments

The connection radius and channel delta experiments above are repeated using both star and ring connection topologies. To simplify the graphs results are only presented for the calendar queue, CCT, and fixed schedule algorithms. As mentioned previously, the behaviour of the CEL algorithms is approximately constant in both connection radius and channel delta. The controlled variables for each of the experiments can be found in Table 5.1.

Channel Delta

In the first experiment, the connection radius is fixed at 4, the event density is fixed at 4, and the channel delta is varied between 0.015625 and 4.0. The experiment is conducted using a connection radius of 4 since for R=1, the ring and star topologies are identical. A separate curve is plotted for execution of each algorithm on each of the topologies (ring and star). The graphs in Figures 5.11 and 5.12 compare the effect of varying channel delta under both the ring and star topologies.

Figure 5.11A plots model level cache behaviour against the channel delta. In terms of model level cache behaviour, the topology of the model had no effect on the calendar queue algorithm, it had a very small effect on the CCT algorithm, and only a slightly larger effect on the fixed schedule algorithm. For a ring topology, the model level cache behaviour of the fixed schedule algorithm is slightly better than that of CCT (particularly for small channel deltas). For the star topology CCT has consistently better cache performance than the fixed schedule.

The kernel level cache behaviour is plotted in Figure 5.11B. Kernel level cache behaviour showed sensitivity to model topology, particularly for the fixed schedule algorithm. The cache performance was close to 2 times worse in a star connected model than in a ring connected model with the same channel delta. The cache performance of CCT was also affected by model topology but to a much lesser degree. Again, performance of the calendar queue algorithm appears independent of model topology. For the star topology, the kernel level cache behaviour of CCT is better than that of the fixed schedule algorithm; for the ring topology, the fixed schedule is slightly better.

Figure 5.11C plots the aggregate cache behaviour versus channel delta. Comments regarding the kernel level cache behaviour apply to the aggregate cache behaviour of the algorithms as well. Topology has no effect on the calendar queue, a small effect on the performance of CCT algorithm, and a significant effect on the fixed schedule

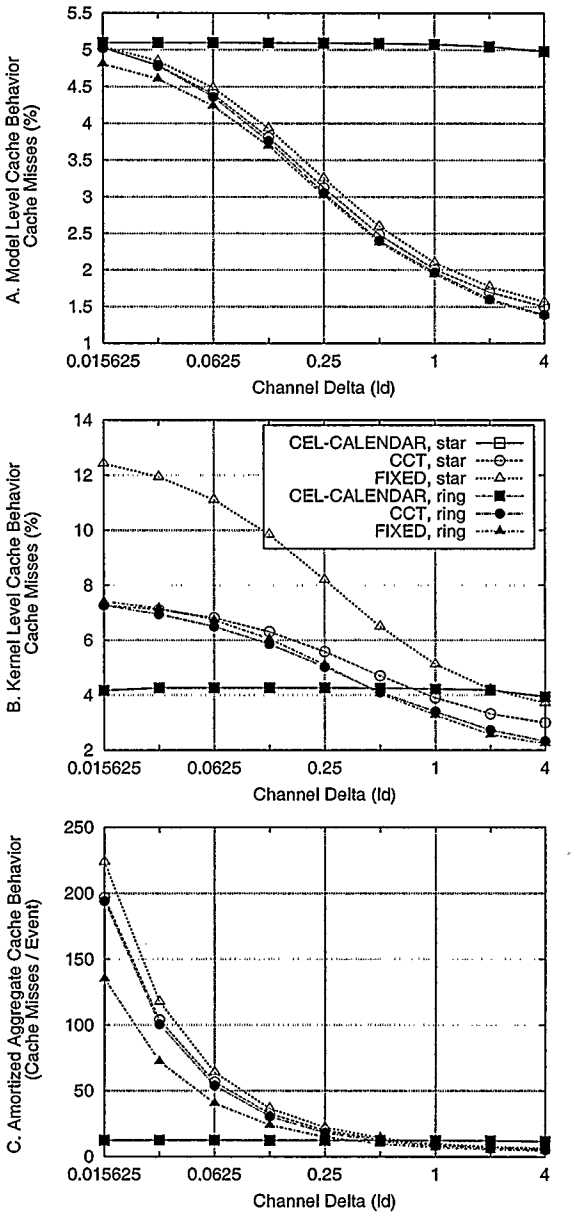


Figure 5.11: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus channel delta.

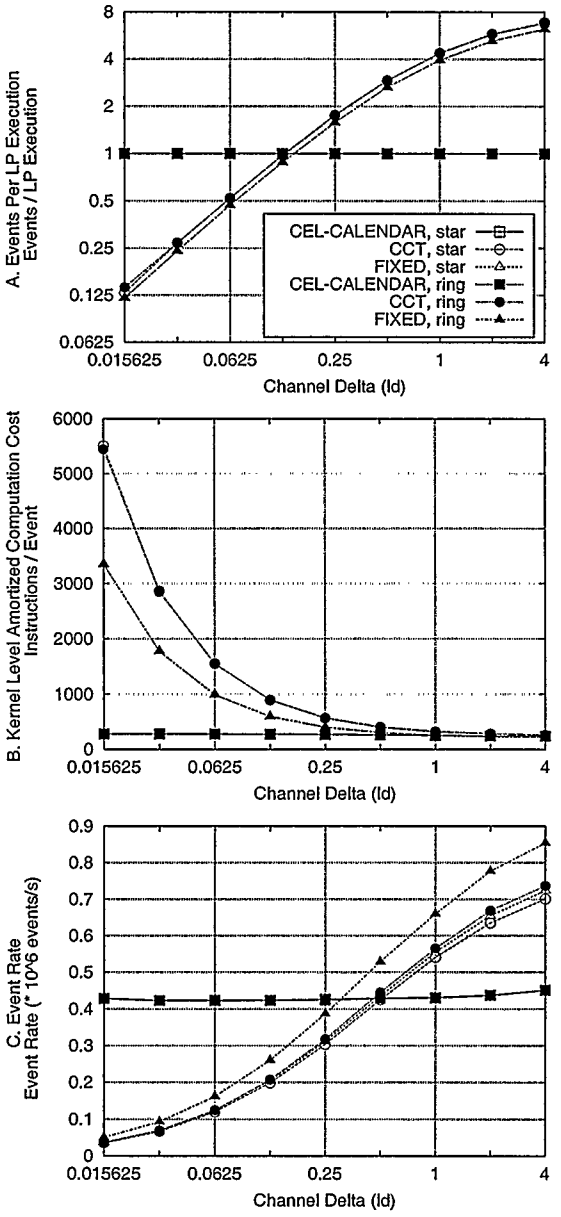


Figure 5.12: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus channel delta.

algorithm. The fixed schedule algorithm experiences between 30% and 40% more cache misses processing a star topology than a ring topology with the same channel delta. The affect of topology on cache is more evident for smaller channel deltas. For the star topology, the aggregate cache behaviour of CCT is better than that of the fixed schedule algorithm; for the ring topology, the fixed schedule is better.

As can be seen in Figure 5.12A, topology has almost no effect on the average number of events executed per LP execution session regardless of the algorithm. The issue is that all of the LPs in the system end up synchronized to the same clock value, so connecting to LPs other than your immediate successor and predecessor makes no difference in terms of the available lookahead. The reason LPs are synchronized to the same clock value is due to the same lookahead being available on each channel (all channel deltas are the same). If LPs further away in the ring were at different times then we might have seen a decreased window of execution (number of events per LP execution). CCT would probably overcome this scenario better than fixed since it would schedule LPs. A better experiment would have used randomly generated channel deltas, so that LPs could actually execute to different times.

In Figure 5.12B, kernel level computation cost is plotted against channel delta. Topology has no observable effect on the kernel level computation cost regardless of the algorithm. This makes sense for the conservative algorithms as there was almost no dependence on topology for the number of events per LP execution. If there was greater variation in the number of events per LP execution, then we might expect a variation in the kernel level computation cost as well.

Figure 5.12C plots the event rate metric versus channel delta. The event rate metric echoes the effect observed in cache behaviour. The conclusions are the same, calendar queue is unaffected by the choice of topology, CCT is somewhat dependent on choice topology, and fixed schedule shows the greatest dependence on topology. The CCT algorithm outperforms the fixed schedule algorithm for the star topology, whereas the fixed schedule outperforms CCT for the ring topology. This behaviour

is due almost entirely to kernel level cache behaviour. This is explained by the kernel level cache behaviour in Figure 5.11C. Although topology does not affect the kernel level computation cost, it does affect the cache behaviour.

Connection Radius

For the second experiment, the channel delta is fixed at 1.0 unit of simulation time and event density is set to 4, while the connection radius is varied between 1 and 32. Again a separate curve is plotted for the execution of each algorithm on each of the topologies. Performance metrics are graphed in Figure 5.13 and 5.14.

The model level cache behaviour (Figure 5.13A) of the calendar queue is unaffected by connection topology. In terms of model level cache behaviour, the affect of topology on the CCT algorithm is negligible, while its affect on the fixed schedule algorithm is at least observable. The larger the connection radius, the greater affect the connection topology has on the fixed schedule algorithm. The ring topology is handled more efficiently than the star topology by the fixed schedule algorithm.

Kernel level cache behaviour is plotted in Figure 5.13B. Again the calendar queue is unaffected by choice of connection topology. In terms of kernel level cache behaviour, both the CCT and fixed schedule algorithms show sensitivity to the topology. The fixed schedule algorithm exhibits consistently greater kernel level cache misses simulating the star model than the corresponding ring model. Up to twice as many cache misses were observed for the star topology than the ring topology when the connection radius was 16 or greater. The fixed schedule algorithm executes LPs executes each LP in the ring in a clockwise fashion. The execution of LP x is followed by the execution of LP $x + 1$, eventually wrapping back around to LP x. This implies that the execution of each LP is preceded and followed by the execution of its closest neighbours. In the ring connection topology the fixed schedule benefits because neighboring LPs are brought into the cache prior to execution. In the star topology, each LP accesses other LPs that are further separated from itself in terms

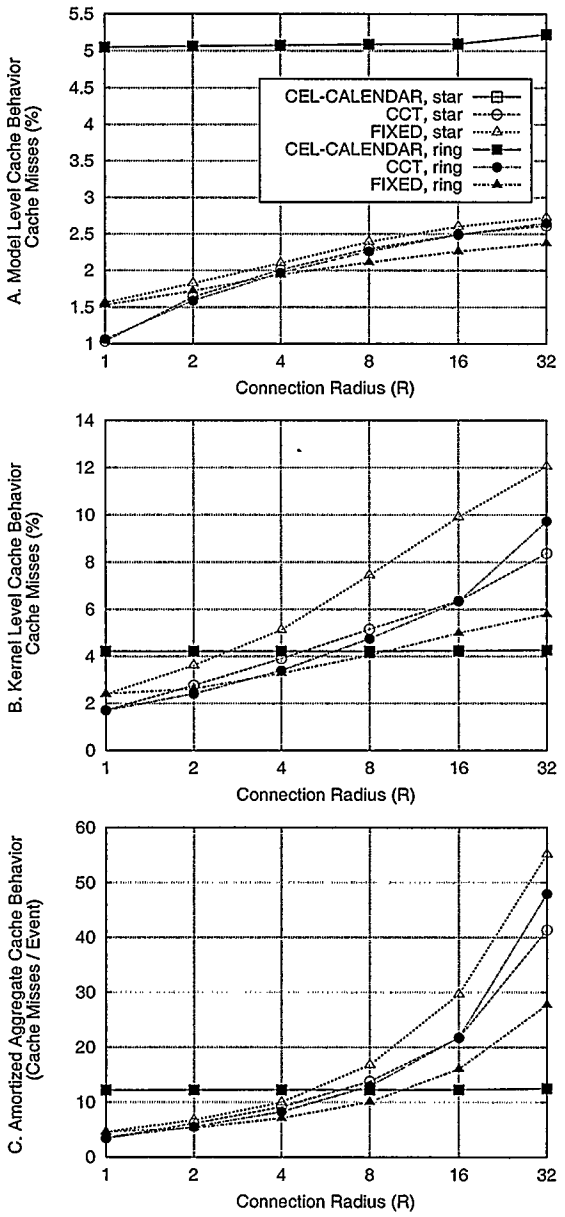


Figure 5.13: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour versus connection radius.

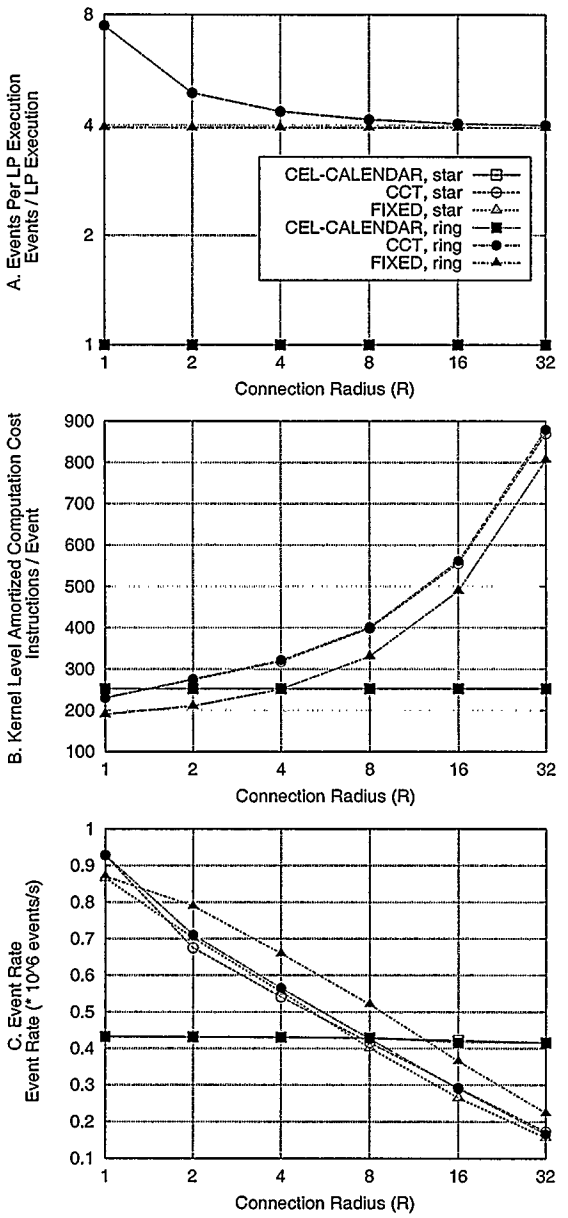


Figure 5.14: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate versus connection radius.

of execution order, i.e., for a given LP, the LPs that must be accessed are not those preceding or following the execution of that LP. The variation in CCT's kernel level cache behaviour is smaller, but more complex. For smaller connection radius the kernel level cache behaviour of CCT is better for the ring topology than for the star topology, but around connection radius 16, the performance of CCT becomes better for the star topology. CCT does not execute the LPs in a fixed order, the order can change and it depends on how critical channels are set. Regardless of topology, it is equally likely that CCT schedules any of its neighbours regardless of where they are in the ring. This implies that the kernel level caching behaviour is similar for both topologies.

The amortized aggregate cache behaviour graph (Figure 5.13C) shows the superiority of the fixed schedule algorithm for the ring connection topology, and the superiority of CCT for the star topology. However, extrapolating from the graph it appears that for a larger connection radius CCT may actually experience more cache misses than the fixed schedule algorithm, independent of topology.

The plots of events per LP execution (Figure 5.14A) show no dependence on topology for any of the three algorithms. As discussed in the previous topology experiment where channel delta was manipulated, it is believed that the events per LP execution are unaffected by topology because the clocks of all LPs are synchronized to the same simulation time. It doesn't matter if the neighbours that an LP is connected to are close by, or far away in the ring they have the same local simulation times and the same lookahead.

The plots of kernel level amortized computation cost (Figure 5.14B) show very little dependence on the choice of topology, for any of the algorithms. Had there been a greater variation in events per LP execution, then greater variation could have been expected in the computation cost also.

The event rate metric is plotted in Figure 5.14C. This graph confirms what was observed in the previous metrics. The calendar queue algorithm's performance is

independent of the topology. The CCT algorithm is less sensitive to the choice of topology than the fixed schedule algorithm, but fails to achieve significantly greater performance. The fixed schedule algorithm shows the greatest dependence on topology, but performs better than CCT in the case of ring topology and comparable in the case of the star topology.

5.3 Model Characteristics Experiments

There are many other model characteristics that could affect algorithm performance. This section will consider a few more model parameters, that ultimately result in the described synthetic model being quite general and comparable to many real world systems and scenarios.

This group of tests will examine the remaining model parameters and study how they affect the performance of the different simulation algorithms. The first experiment considers computation grain, which corresponds to the amount of work required to process an event. The second experiment will compare the performance of the different algorithms when the amount of LP state is varied. The third experiment manipulates the timestamp increment distribution to observe the effect on algorithm performance.

5.3.1 Computation Grain Experiment

This experiment will involve varying the amount of work done to process an event at the model level. If the processing of an event by the model level code is simple, and quick, this corresponds to a small computation grain. If the processing of an event at the model level is complicated or time consuming this corresponds to a large computation grain. Computation grain experiments are not particularly relevant to testing sequential simulation kernels. Their applicability is in parallel simulation where busy cycles of one cpu could affect the arrival of events at another cpu. Also,

for parallel simulation a large computation grain provides opportunity for parallelism. This test demonstrates that choosing an efficient algorithm is only relevant if the model has been described in a manner such that the computation grain does not nullify the benefits of the different algorithms. Graphs are only presented for the event rate metric, since the behaviour of the other metrics is essentially constant in terms of additional computation grain. Cache behaviour and kernel level computation costs are unaffected by additional computation grain.

Two different computation grain experiments were conducted. In experiment 1, the number of LPs is fixed at 8192, the connection radius is 4, and the event density is 4. In experiment 2, the corresponding model parameters were 16384 LPs, connection radius of 1, and event density of 8. The controlled experiment parameters are in Table 5.1.

Consider the computation grain of the synthetic workload that is already present. The synthetic workload has an inherent computation grain which corresponds to execution of the model level code, the computation grain that is manipulated in these experiments is in addition to this inherent computation grain. It is possible to estimate the inherent computation grain for the synthetic workload; however, equating computation grain with a measurement of time is error prone, since the time could change depending on the performance of the algorithm. All model parameters come into play so the tolerable amount computation grain may vary widely between different simulations. The inherent computation grain is estimated by taking the event rate and inverting it, this gives the time to execute a single event. Multiply this by the ratio of model level instructions to total instructions to obtain the time spent per event in model level code on average (i.e., the inherent computation grain). Multiply this by 1e6 to get the time in microseconds. Using time to measure the additional computation is not as error prone, since the time spent in the busy wait loop which simulates the additional computation grain, does not depend on algorithm performance. Table 5.2 shows the estimated inherent computation grain (T_{MLPE}), the time

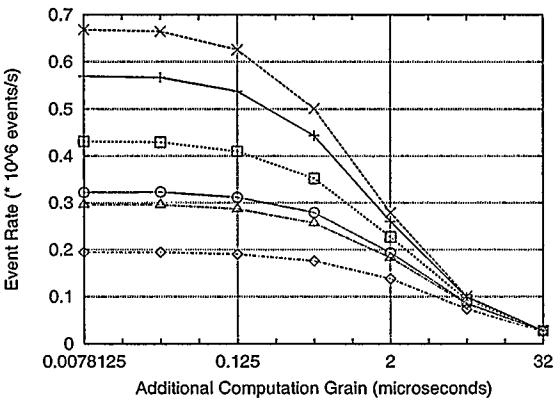
Experiment 1 (N8192, R4, D4)			
Algorithm	$T(\mu s)$	$T_{KLPE}(\mu s)$	$T_{MLPE}(\mu s)$
CEL-CALENDAR	2.32	1.10 (47.34%)	1.22 (52.66%)
CEL-SPLAY	3.10	1.08 (34.82%)	2.02 (65.18%)
CEL-HO_HEAP	3.34	0.90 (26.84%)	2.45 (73.16%)
CEL-HENRIKSEN	5.20	1.32 (25.41%)	3.88 (74.59%)
CCT-FIFO	1.77	0.73 (41.17%)	1.04 (58.83%)
FIXED-FIFO	1.52	0.72 (47.34%)	0.80 (52.66%)
Experiment 2 (N16384, R1, D8)			
Algorithm	$T(\mu s)$	$T_{KLPE}(\mu s)$	$T_{MLPE}(\mu s)$
CEL-CALENDAR	2.40	1.15 (47.66%)	1.26 (52.34%)
CEL-SPLAY	3.70	1.24 (33.45%)	2.46 (66.55%)
CEL-HO_HEAP	4.41	1.09 (24.69%)	3.32 (75.31%)
CEL-HENRIKSEN	7.64	1.59 (20.80%)	6.05 (79.20%)
CCT-FIFO	0.91	0.48 (52.42%)	0.43 (47.58%)
FIXED-FIFO	0.94	0.52 (55.42%)	0.42 (44.58%)

Table 5.2: Inherent Computation Grain (in microseconds)

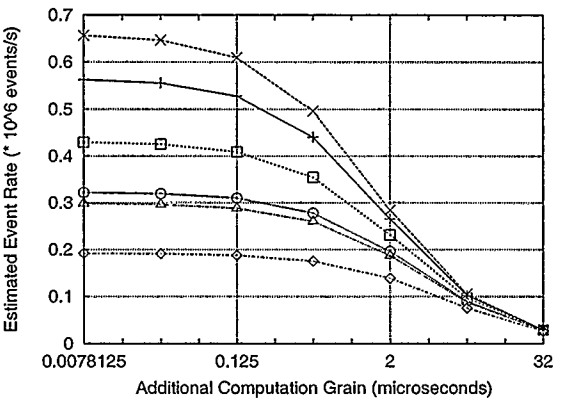
spent per event in kernel code per event (T_{KLPE}), and the total time spent in the simulator per event (T).

As the additional computation grain is increased the difference between the algorithms becomes rapidly smaller, as can be seen in Figure 5.15. For simulated computation grains of less than 1 microsecond there are substantial differences between the algorithms, but for larger computation grains, i.e., 4 microseconds or greater, the choice of algorithm is almost irrelevant. The results of this experiment confirm the larger the computation grain, the less the kernel algorithm will impact performance.

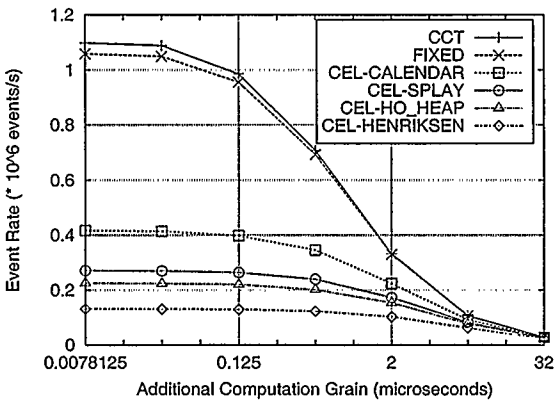
The rapid decrease in event rate was surprising at first, but this does make sense



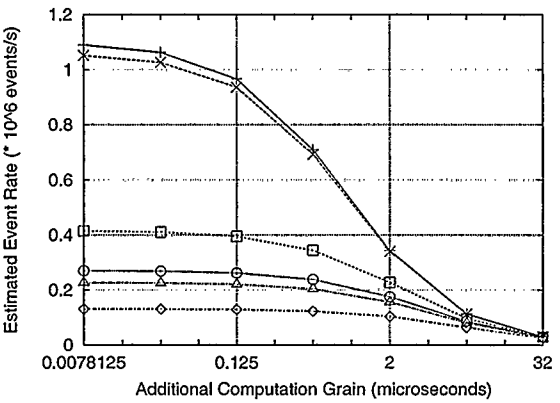
Exp. 1 Simulation Results



Exp. 1 Analytical Estimation



Exp. 2 Simulation Results



Exp. 2 Analytical Estimation

Figure 5.15: Computation Grain Experiment

since as the following derivation will show, event rate is a rational function of $1/G$.

V	average Event Rate during simulation
M	Total Events executed during simulation
T	Total Time required for the simulation
T_k	Total Time spent in the simulation kernel level code
T_m	Total Time spent in model level code
T_{KLPE}	Time spent in kernel level code per event (on average)
T_{MLPE}	Time spent in model level code per event (on average)
G	Additional computational grain to be simulated

$$
\begin{aligned}
T &= T_k + T_m \\
T_k &= T_{KLPE} \times M \\
T_m &= T_{MLPE} \times M \\
\hline
V &= \frac{M}{T} \\
&= \frac{M}{T_k + T_m} \\
&= \frac{M}{T_{KLPE} \times M + T_{MLPE} \times M} \\
&= \frac{1}{T_{KLPE} + T_{MLPE}}
\end{aligned}
$$

T_{KLPE} is constant in G, so we have a rational function of T_{MLPE}. In fact, given the event rate of the simulation where $G = 0$, the event rate for any given value of G can actually be predicted with the formula $V(G) = \frac{1}{1/V(0)+G}$. The second column of graphs in Figure 5.15 demonstrates the use of the analytical formula for predicting the event rate of the different algorithms.

5.3.2 LP State Size Experiment

The purpose of this experiment is to study the effect of LP state size on the performance of the different algorithms. It is anticipated that the effect of changing the LP state size will be noticed primarily in the model level cache behaviour. This experiment is included to highlight the model level cache advantages of the conservative algorithms.

The model parameter SS is used to specify the number of elements in the LP state array. Each time an event is executed the entire array is examined and the average of the elements is computed. The problem with modeling LP state in this manner is that although the model level cache performance is affected, the computation grain is also significantly increased. This is due to running a loop to examine each element in the array. To reduce this affect, each element in the array is cache aligned and padded to the length of a cache line. This ensures that the greatest amount of memory can be accessed using the fewest possible instructions. This will highlight the affects of cache behaviour while minimizing the increase in computation grain. In the case of the conservative algorithms it is expected that the values in the array will be cached and reused approximately as many times as the event density parameter (D). The CEL algorithms will jump back and forth between LPs and as a result may need to reload the array each time an event is executed.

The LP state size experiments fix the number of LPs at 8192. Two experiments are conducted: one with event density 1, one with event density 4. In each experiment, the standard set of algorithms are compared as the LP state size is varied between 1 and 32 items. Each item corresponds to a cache aligned chunk of memory 32 bytes long, the size of one cache line. Table 5.1 contains the remaining controlled parameters for this experiment. Since varying the LP state size had no observable effect on kernel level cache behaviour, events per LP execution, and kernel level computation cost, only the model level cache behaviour, aggregate cache behaviour, and event rate metrics are presented. See Figures 5.16 and 5.17 for the results.

Figures 5.16A and 5.17A plot the percentage of cache misses as LP state size is increased from 1 to 32. As LP state size increases, the cache miss percentage is increasing approximately linearly for both CEL and conservative algorithms. The conservative algorithms are able to maintain a lower percentage of model level cache misses than the CEL algorithms. The CEL algorithms show a significant dependence on the LP state size, reaching 18% for models with larger LP state sizes, this is about

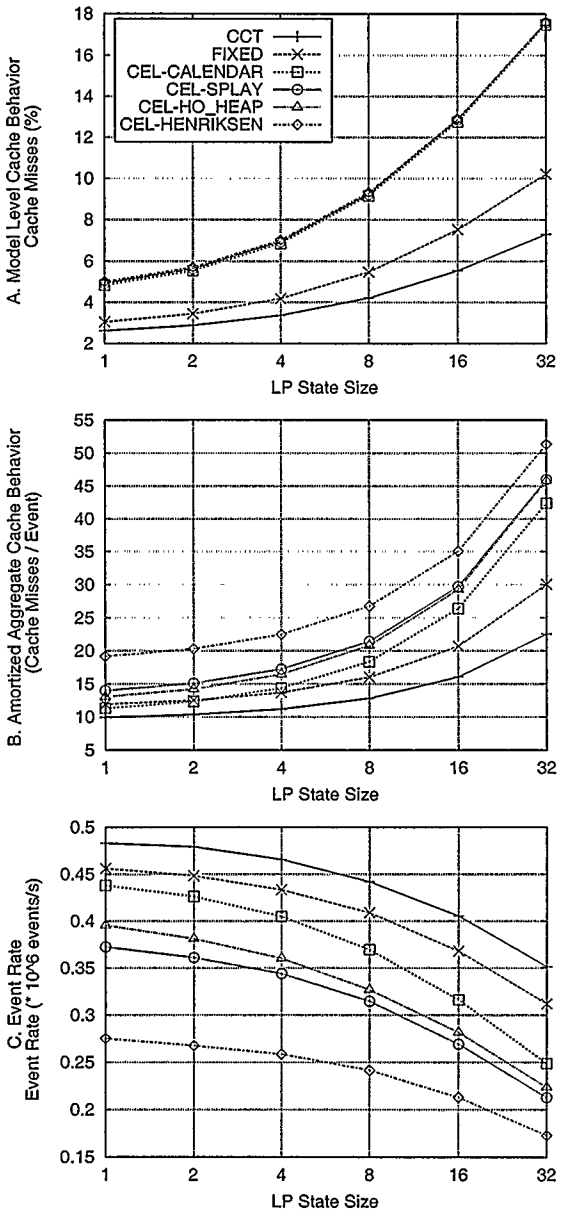


Figure 5.16: Plots of A. Model Level Cache Behaviour, B. Amortized Aggregate Cache Behaviour and C. Event Rate versus state size, for D = 1.

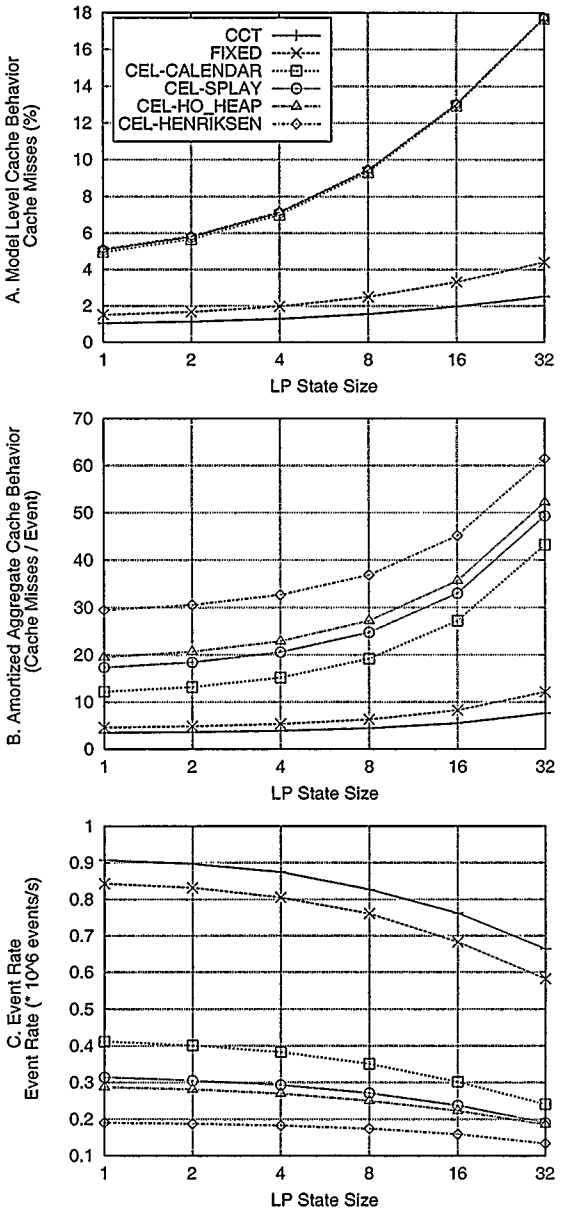


Figure 5.17: Plots of A. Model Level Cache Behaviour, B. Amortized Aggregate Cache Behaviour and C. Event Rate versus state size, for D = 4.

9× worse cache performance than the CCT algorithm (in the case where event density = 4). For model level cache behaviour the variation between the CEL algorithms themselves is altogether absent, while there is some difference observed between the fixed schedule and CCT algorithm. Comparing Figures 5.16A and 5.17A, it is observed that for CEL algorithms, the model level cache behaviour is unaffected by the event density parameter; the cache performance does not depend on event density. For conservative algorithms the sensitivity to state size is negatively correlated with the event density. The higher the event density, the less affect larger state sizes have on cache behaviour.

The graph of amortized aggregate cache behaviour (Figures 5.16B and 5.17B) illustrates the same behaviours observed in model level cache behaviour. The CEL algorithms are spread out over a wider range but the shape and slope of the curves is the same. In Figure 5.17B, Henriksen's algorithm experiences about 6× as many cache misses as the conservative algorithms for an LP state size of 32. The Calendar queue has better cache performance but still results show about 4× more cache misses than the conservative algorithms. CCT performs around 30% better than the fixed schedule algorithm in terms of amortized aggregate cache behaviour when event density is 1, but this is less pronounced for event density 4 (Figure 5.17B).

The Event Rate graphs (Figures 5.16C and 5.17C) show the superiority of the conservative algorithms. Although the absolute event rate is decreasing on account of the increase in computation grain, the relative performance of the conservative algorithms continues to improve. It is possible to infer that this relative improvement is due to better cache performance since the kernel level computation cost metric (not shown) is constant in the LP state size, and the model level computation cost is the same, independent of the algorithm.

5.3.3 Timestamp Increment Distribution Experiment

The purpose of this set of experiments is to determine the extent to which the timestamp increment distribution affects the performance of the different algorithms. This experiment will compare the performance of the algorithms for six timestamp increment distributions: constant, exponential, uniform, biased, bi-modal, and triangular (See table 4.1). Six models were selected to illustrate the range of behaviours of the algorithms: (N8192, D4, R32, L1), (N8192, D0.25, R1, L1), (N8192, D4, R1, L0.125), (N8192, D4, R1, L1), (N8192, D4, R1, L2), (N8192, D32, R1, L1). For each model, each algorithm was tested using each of the six timestamp increment distributions.

Table 5.3 shows the results obtained for the first model (N8192, D4, R32, L1), results for the other models are available in appendix A. The data collected for all models is presented in Figures 5.18 and 5.19. Vertical columns of the graphs separate the different algorithms, the linestyle of the individual markers distinguishes between the model topologies, and finally the markers themselves indicate the minimum, maximum, average, and standard deviation of the different distributions with regards to a particular metric. Within a column the six markers from left to right correspond to the six model topologies listed top to bottom in the legend. The marker is a rectangular box, limited vertically by the average +/- one standard deviation. A vertical line segment extends from the top of the box to the maximum value, a second line segment extends from the bottom of the box to the minimum value. The average is indicated by a horizontal bar in the middle of the box.

Looking first at the model level cache behaviour (Figure 5.18A), it can be seen that the timestamp increment distribution has a significant effect on all of the CEL algorithms. For any given model the percentage of cache misses observed ranges about 1.0-1.5%. For example, refering to Table 5.3 it was observed for CEL algorithms that using the exponential distribution resulted in approximately 4.16% cache misses, while the uniform distribution saw approximately 5.27% for the model (N8192, D4, R32, L1). The affect of the timestamp increment distribution on model level cache

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behaviour Cache Misses (%)								
CEL-CALENDAR	3.88	4.12	5.23	5.19	4.79	5.26	4.75	0.60
CEL-SPLAY	3.89	4.16	5.27	5.19	4.83	5.24	4.76	0.60
CEL-HO_HEAP	3.92	4.17	5.29	5.24	4.85	5.27	4.79	0.60
CEL-HENRIKSEN	3.88	4.18	5.29	5.20	4.85	5.27	4.78	0.61
CCT-FIFO	2.36	2.11	2.65	2.63	2.56	2.64	2.49	0.22
FIXED-FIFO	2.06	1.89	2.37	2.36	2.31	2.37	2.23	0.20
Kernel Level Cache Behaviour Cache Misses (%)								
CEL-CALENDAR	1.65	4.12	4.27	4.77	4.19	7.20	4.37	1.77
CEL-SPLAY	2.88	5.35	5.11	3.66	5.00	4.74	4.46	0.97
CEL-HO_HEAP	3.64	4.51	4.62	4.70	4.46	4.70	4.44	0.40
CEL-HENRIKSEN	1.90	8.74	10.42	4.59	9.44	10.49	7.60	3.54
CCT-FIFO	8.02	9.67	9.73	9.71	10.06	9.71	9.48	0.73
FIXED-FIFO	4.66	5.73	5.79	5.77	5.97	5.77	5.61	0.47
Amortized Aggregate Cache Behaviour Cache Misses / Event								
CEL-CALENDAR	6.43	12.27	12.47	13.59	12.28	21.94	13.16	4.99
CEL-SPLAY	8.25	17.80	17.08	13.01	16.63	16.12	14.82	3.62
CEL-HO_HEAP	10.91	18.62	18.94	19.27	18.30	19.24	17.55	3.27
CEL-HENRIKSEN	7.06	24.59	30.39	14.42	27.46	31.40	22.55	9.74
CCT-FIFO	24.81	47.88	47.95	47.91	57.00	47.92	45.58	10.80
FIXED-FIFO	14.44	27.58	27.72	27.67	32.68	27.65	26.29	6.14
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	7.96	4.00	3.99	3.98	3.33	3.99	4.54	1.70
FIXED-FIFO	7.88	3.95	3.94	3.94	3.29	3.94	4.49	1.68
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	235.09	243.79	252.23	266.99	240.94	624.69	310.62	154.26
CEL-SPLAY	248.65	428.77	421.69	360.86	413.28	413.21	381.08	69.19
CEL-HO_HEAP	385.55	609.43	612.12	618.02	602.74	616.54	574.07	92.52
CEL-HENRIKSEN	398.37	576.33	659.13	462.38	635.41	689.33	570.16	116.23
CCT-FIFO	513.15	883.03	879.05	879.31	1027.40	880.19	843.69	172.28
FIXED-FIFO	476.21	809.35	805.90	806.09	938.98	806.26	773.80	155.07
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.67	0.40	0.42	0.38	0.41	0.22	0.41	0.14
CEL-SPLAY	0.63	0.30	0.32	0.41	0.32	0.33	0.39	0.13
CEL-HO_HEAP	0.46	0.29	0.30	0.29	0.30	0.28	0.32	0.07
CEL-HENRIKSEN	0.62	0.21	0.18	0.34	0.19	0.17	0.29	0.18
CCT-FIFO	0.29	0.16	0.17	0.17	0.14	0.16	0.18	0.06
FIXED-FIFO	0.39	0.22	0.22	0.22	0.19	0.22	0.25	0.07

Table 5.3: Results for D4_R32_L1 Distribution experiment

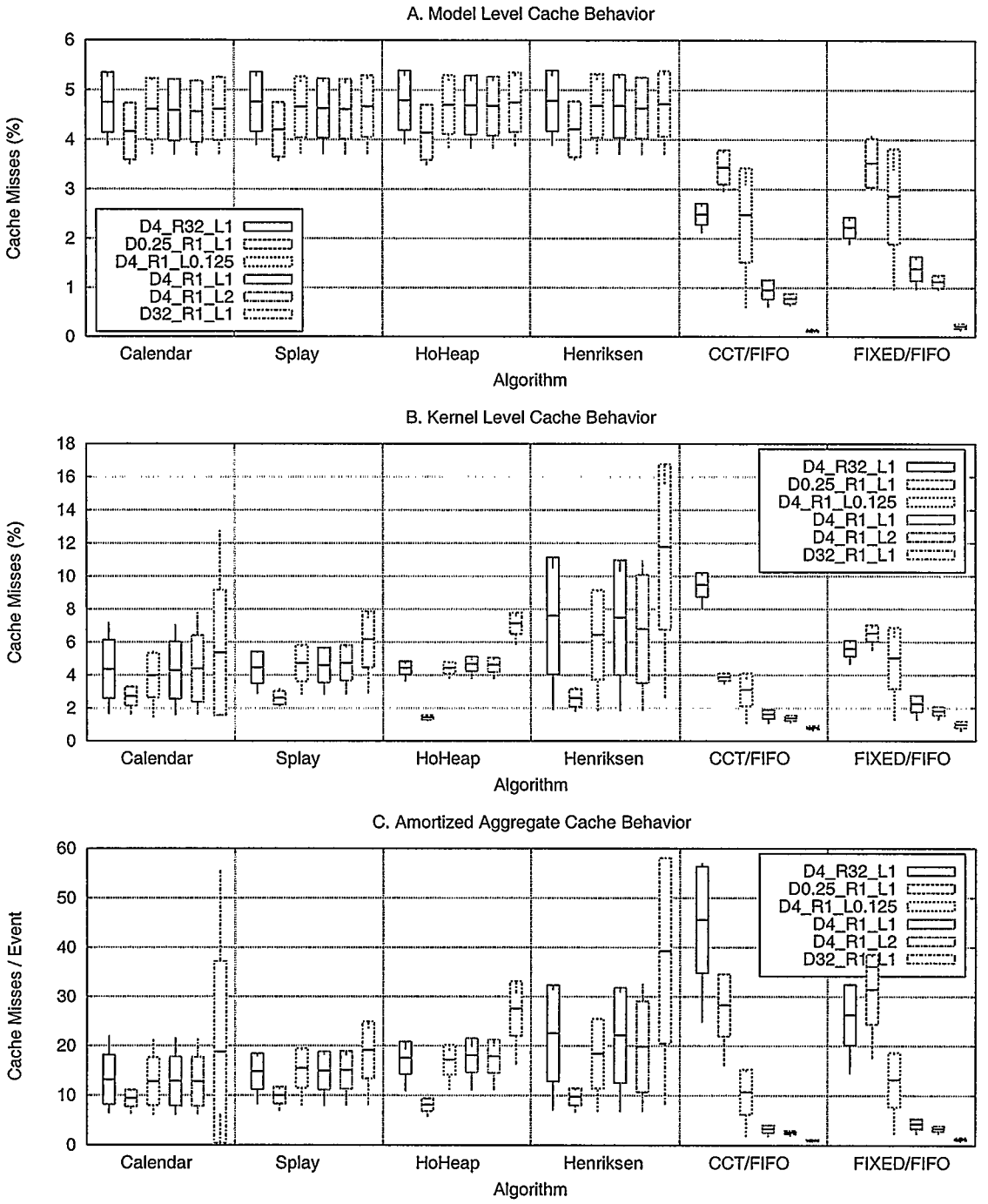


Figure 5.18: Plots of A. Model Level Cache Behaviour, B. Kernel Level Cache Behaviour and C. Aggregate Cache Behaviour for different algorithms, models and timestamp increment distributions.

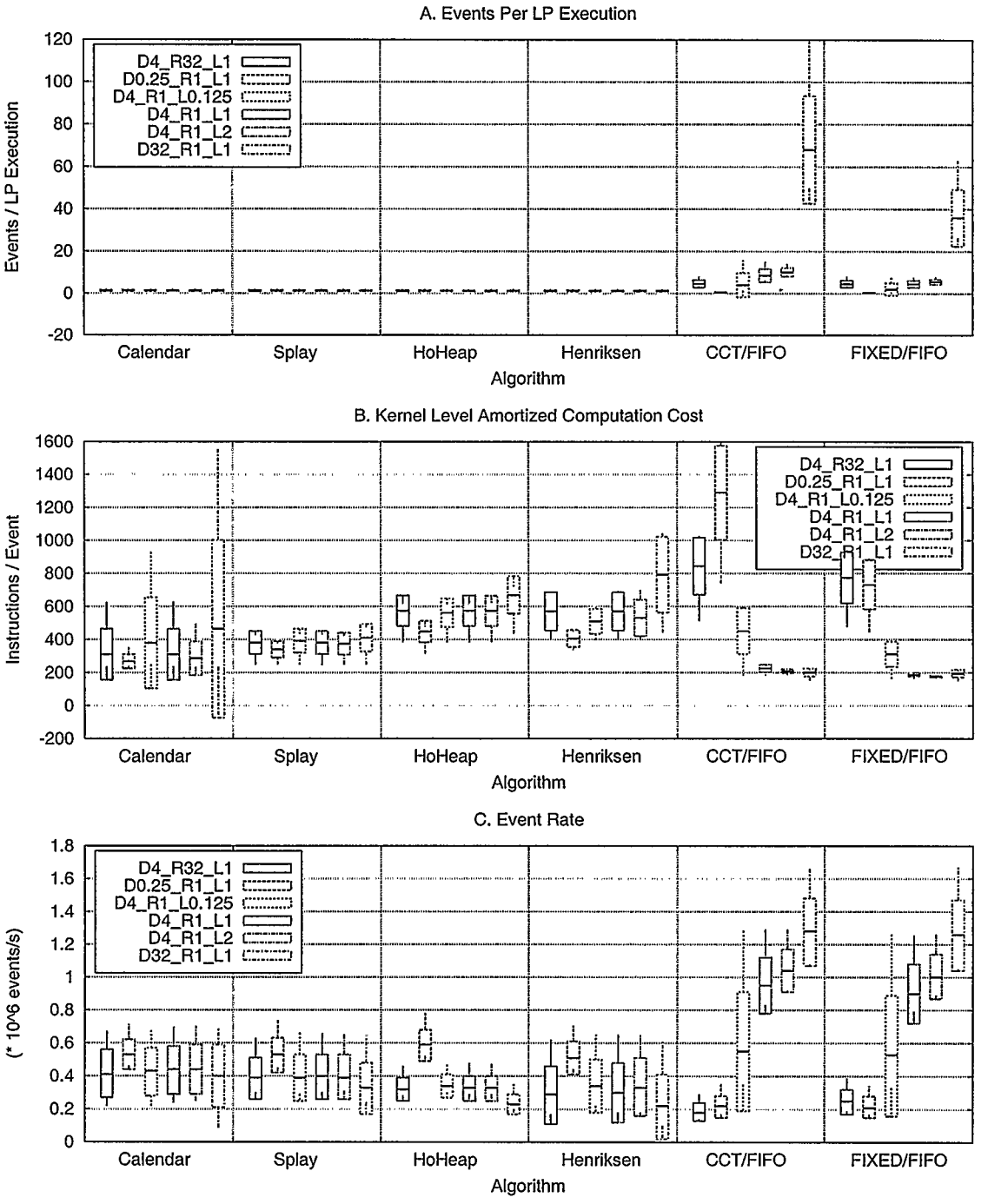


Figure 5.19: Plots of A. Events per LP Execution, B. Kernel Level Amortized Computation Cost and C. Event Rate for different algorithms, models and timestamp increment distributions.

behaviour seems about the same regardless of CEL algorithm. For the conservative algorithms, slightly less variation was observed within a given model, except for model (N8192, D4, R1, L0.125) where significantly greater variation was observed. Less variation suggests less sensitivity to the timestamp increment distribution, at least in models with large channel delta that avoid low lookahead cycles. The key observation is that in most cases, the conservative algorithms show less sensitivity to the timestamp increment distribution in terms of their model level cache behaviour.

The CEL algorithms can be roughly ranked according to their kernel level cache behaviours' (Figure 5.18B) sensitivity to the timestamp increment distribution. From most sensitive to least sensitive: Henriksen, Calendar, Splay, and Ho_Heap. The conservative algorithms are again less sensitive to timestamp increment distribution, except in cases of low lookahead. Kernel level cache behaviour appears extremely sensitive to distribution for calendar queue and Henriksen's algorithm when the event density is large (or possibly when the event population is large), as evidenced by model (N8192, D32, R1, L1).

For the CEL algorithms, observations made regarding kernel level cache behaviour apply also to amortized aggregate cache behaviour (Figure 5.18C). For the conservative algorithms, aggregate cache behaviour exhibits variation comparable to CEL algorithms for models with low lookahead ((N8192, D0.25, R1, L1), (N8192, D4, R1, L0.125)) or high connectivity (N8192, D4, R32, L1). For the other models, conservative algorithms show far less dependence on the timestamp increment distribution than the CEL algorithms do. Another interesting observation is that CCT shows greater sensitivity to distribution when connection radius is high, while fixed shows greater sensitivity when event density is low.

The number of events per LP execution (Figure 5.19A) shows no sensitivity to the timestamp increment distribution for CEL algorithms. The CEL algorithms achieve 2 events per LP execution for the constant distribution because the increment was constant at 1.0. For the conservative algorithms, there is very little dependence on

timestamp increment distribution when event density is small (N8192, D0.25, R1, L1) and significant dependence when event density is large (N8192, D32, R1, L1). Dependence on timestamp increment distribution for conservative algorithms seems to be proportional to some function of event density. When event density is large, CCT shows greater sensitivity to distribution than the fixed schedule.

Figure 5.19B plots the kernel level computation cost. The splay, ho_heap, and Henriksen's algorithms show moderate variation in their kernel level computation cost as timestamp increment distribution is varied. The calendar queue seems particularly sensitive to timestamp increment distribution for models (N8192, D4, R1, L0.125), (N8192, D32, R1, L1). For the conservative algorithms, again greater sensitivity to scheduling distribution is observed in cases of low lookahead ((N8192, D0.25, R1, L1), (N8192, D4, R1, L0.125)) or high connectivity (N8192, D4, R32, L1). The fixed schedule is less sensitive to timestamp increment distribution than CCT in terms of kernel level computation cost.

When looking at the event rate metric it is important to keep in mind that this reflects the inherent computation grain of the simulation. Variation in event rate between different distributions for the same algorithm could be due in part to the expenses associated with generating different random variates. For example, generating a pseudo random number that fits a normal or uniform distribution is much less expensive than generating one that fits an exponential distribution. Computation grain aside, several observations can be made regarding the variation in event rate across different timestamp increment distributions.

The event rate metric is plotted in Figure 5.19C. For most models, the CEL algorithms ranked according to increasing variation in event rate are ho_heap, splay, calendar, and Henriksen's. The conservative algorithms exhibit less absolute variation in event rate than the CEL algorithms in models ((N8192, D4, R32, L1), (N8192, D0.25, R1, L1), (N8192, D4, R1, L2)). They exhibit greater variation in models ((N8192, D4, R1, L1), (N8192, D4, R1, L0.125), (N8192, D32, R1, L1)). The relative

variation as compared to the average event rate, is less for conservative algorithms than CEL-based algorithms in all but two models, (N8192, D4, R1, L0.125) and (N8192, D0.25, R1, L1) where available lookahead was not always adequate to execute an event. These are situations in which the conservative algorithms achieved less than 1 event per LP execution. The variation in event rate observed for CCT and the fixed schedule algorithm are very close.

5.4 Summary

This chapter has presented the results of thesis research conducted. These experiments demonstrate the range of performance possible using the tested simulation algorithms. Most experiments manipulated a single model parameter while holding other model parameters constant. The parameters of the synthetic workload model that were manipulated were: number of LPs, event density, connection radius, channel delta, connection topology (ring and star), computation grain, LP state size, and timestamp increment distribution.

The performance of the conservative algorithms decreases as connectivity is increased and eventually becomes worse than that of the CEL algorithms. This is due to increased channel scanning costs that result in increased instruction cost and worse cache behaviour. The cache behaviour is negatively affected for higher connectivity because more state is accessed. CEL algorithms are insensitive to changes in the connection radius of the model.

The average number of events per LP execution is influenced by the event density and the lookahead. Increasing event density results in more events per LP execution session for the conservative algorithms resulting in better cache behaviour and reducing the frequency of accessing the LP scheduling queue. The conservative algorithms achieved event rates of over 5 times that of a heap CEL algorithm, and up to nearly 3 times that of a calendar queue CEL algorithm. The L2 cache miss rate was up to 18

times lower than that for the splay tree CEL algorithm and up to 12 times lower than that for the calendar queue algorithm. However, with low event density an event rate up to 2.5 times less than that of CEL algorithms was observed.

Increasing the lookahead also results in more events per LP execution session giving better cache behaviour and reducing the LP scheduling cost. Although the conservative algorithms achieved an event rate over 3 times that of heap and splay tree CEL algorithms at high lookahead, an event rate 4 times lower than that of the calendar queue CEL algorithm was observed at low lookahead. As lookahead decreases the temporal separation of events becomes greater than the lookahead resulting in low lookahead cycles and in turn many LP execution sessions in which no events are executed.

The choice of model topology, either ring or star, did not have much affect on the performance of the various algorithms. The computation grain experiment serves as a reminder that the larger the computation grain the less dependence there is on choice of algorithm. When the scheduling distribution was manipulated the relative variation of CMB-based algorithms was in general smaller than for CEL algorithms except when events per LP execution was less than 1.

The behaviour of the calendar queue algorithm was very stable in the experiments conducted. Research conducted by Rönngren *et al.* has shown that the performance of the calendar queue can vary significantly with certain distributions, or when the distribution is changing [32]. Also, because the queue size was not changing during these simulation runs, the calendar queue did not need to perform resizes. If the queue size was dynamic, the calendar queue might not have performed as well. Parallel variations of the up down hold model and the interaction hold model could have perhaps demonstrated these scenarios.

Chapter 6

Summary

Discrete event simulation (DES) is in widespread use as tool for modeling a system as it evolves over time. The majority of simulation studies are conducted sequentially. One reason for this is the availability of single or dual processor computers that can be more cost effective than larger parallel computers. A second reason that simulations are conducted sequentially is that simulation studies often require the execution of thousands of simulation runs. In this situation better throughput is achieved using sequential DES rather than parallel discrete event simulation (PDES) techniques, due to the lower computational efficiency generally achieved by PDES systems.

Most sequential discrete event simulators are based on the central event list (CEL) algorithm which uses a single priority queue to order the execution of all events in the system. Much of sequential DES research has focused on the implementation of the priority queue. This thesis proposes a different approach to sequential DES based on using PDES algorithms in a sequential execution environment.

The critical channel traversing (CCT) algorithm is a PDES algorithm that has demonstrated excellent performance in sequential runs. Using the CCT algorithm, the ATM-TN network simulator was able to achieve an event rate three times greater than when using a splay tree based CEL [41]. The IP-TN network simulator has performed up to four times better using CCT than using a CEL-based simulator

employing a heap [21]. The excellent performance results in these papers motivated the examination of PDES algorithms for potential application to sequential DES.

This thesis has explored the use of channel based conservative PDES mechanisms in a sequential execution environment. The performance of the conservative algorithms was compared with numerous implementations of a CEL-based simulator including Henriksen's, heap, splay tree, and calendar queue. The complexity of channel based conservative algorithms was analyzed and compared to that of various CEL implementations. A parameterized synthetic workload model was developed as a basis for empirical comparison of the different algorithms. Six performance metrics were defined and used to analyze the range of performance possible for each of the model parameters.

It was shown that several channel based conservative algorithms including CCT could under identifiable conditions, achieve significantly better performance than a wide range of CEL-based algorithms. This chapter summarizes the results and concludes with an exploration of future work.

6.1 Conclusions

Table 6.1 summarizes the relative speedup in event rate of the CCT algorithm versus the calendar queue. The calendar queue was the best performing CEL-based algo-

Experiment	Worst	Best
Number of LPs	1.3×	2.3×
Event Density	0.40×	3.0×
Channel Delta	0.06×	2.5×
Connection Radius	0.70×	2.2×
LP State Size	2.2×	2.8×

Table 6.1: Relative Speedup of CCT versus CEL - calendar queue.

rithm in all experiments conducted with the synthetic workload model used in this thesis. The results presented here indicate the best and worst performance observed while manipulating the associated model parameter in the corresponding experiment (see Chapter 5). The relative speedup observed was between 2× and 3× across a wide range of model parameters. However, under adverse model conditions the CCT algorithm was almost 17× slower than the calendar queue.

Channel based conservative PDES algorithms relax the total timestamp ordering normally imposed by CEL-based simulators. This difference allows certain events to be executed out of timestamp order, provided that the events are independent and executed by different LPs. Consider two events A and B with timestamps T_A and T_B, respectively. Event A can be executed before event B even though $T_b < T_a$, if there is enough lookahead between the LPs for which events A and B are scheduled. Relaxing the total timestamp ordering has two benefits: reduced complexity through simplified event scheduling and improved cache behaviour.

When the average number of events processed in an LP execution session is greater than 1, the conservative algorithms can achieve significantly better cache performance than CEL algorithms. The conservative algorithms execute numerous events at the same LP whose state is already cached, while the CEL-based algorithms may need to re-fetch the LP state for each event processed since it will be interleaved with the execution of other events at other LPs. In addition to cache advantages, the LP scheduling queue is sorted less frequently resulting in lower instruction costs. This can occur whenever the events executed per LP execution session is greater than 1.

If there is not enough lookahead in the model, then a low-lookahead cycle may occur. In this situation LPs are scheduled for execution, but they are unable to process their next event. This has a negative impact on cache performance and results in the LP scheduling queue being sorted multiple times to execute a single event. If there is not adequate lookahead in the model such that each LP can execute at least 1 event on average, then performance will be poor relative to CEL-based algorithms.

Channel based conservative algorithms are also sensitive to the connectivity of the model. For the algorithms tested in this thesis, the complexity of the safetime calculation was linear in the number of channels. The performance benefits of a conservative approach are offset when the cost of the safetime calculation becomes too high.

This thesis has demonstrated that channel based conservative algorithms can be an efficient option for sequential simulation under identifiable conditions. The ideal conditions are situations where there is relatively low connectivity of LPs, high event density, and adequate lookahead to ensure at least 1 event is executed per LP execution session. As evidenced by the sequential performance of IP-TN [21] and ATM [41] these conditions are present in many real world simulation problems.

6.2 Future Work

This thesis explored the range of performance possible using a channel based conservative PDES algorithm to simulate a particular synthetic workload model. Future work could include examination and development of new algorithms, as well as comparison of these algorithms under different workloads.

Further experimentation with synthetic workload models could address some of the shortcomings of the hold model used in this thesis. The interaction hold model [27] addresses the interaction of multiple timestamp increment distributions that is common in real world simulation. The up/down hold model [2] addresses the changing queue size also observed in many real world simulations. Experimenting with additional model topologies such as toroid or tandem networks could also prove interesting. In addition, obtaining benchmark results for real world simulations such as network simulation, or process control modeling would provide further evidence to support the use of PDES techniques sequentially.

Optimistic algorithms are not good candidates for sequential execution due to

the large overheads of state saving and rollback. For a sequential simulator, time not spent executing events is time wasted. Sequential time-stepped algorithms or synchronous conservative algorithms would be worth exploring as channel scanning costs are avoided. Variations on the CCT mechanism, such as Receive side CCT [34] might also be able to reduce channel scanning costs. Another approach is to employ a deadlock detection and recovery algorithm such as the one described by Fujimoto [14]. The drawback of a deadlock detection and recovery approach in parallel is that it can result in a large portion of the simulations being deadlocked and thus running almost sequentially. This is not an issue in a sequential execution environment, since simulator execution is already sequential.

A final area of interest is that of cost efficient techniques for overcoming the low-lookahead cycle problem. Many such techniques have been developed for CMB-based algorithms in a parallel environment such as Carrier NULL Messages [40] and Cooperative Acceleration [1]. Optimization of these techniques for a sequential environment could be explored.

Appendix A

Distribution Experiment Results

Algorithm	Distribution						Mean	Standard
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		Deviation
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.88	4.12	5.23	5.19	4.79	5.26	4.75	0.60
CEL-SPLAY	3.89	4.16	5.27	5.19	4.83	5.24	4.76	0.60
CEL-HO_HEAP	3.92	4.17	5.29	5.24	4.85	5.27	4.79	0.60
CEL-HENRIKSEN	3.88	4.18	5.29	5.20	4.85	5.27	4.78	0.61
CCT-FIFO	2.36	2.11	2.65	2.63	2.56	2.64	2.49	0.22
FIXED-FIFO	2.06	1.89	2.37	2.36	2.31	2.37	2.23	0.20
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.65	4.12	4.27	4.77	4.19	7.20	4.37	1.77
CEL-SPLAY	2.88	5.35	5.11	3.66	5.00	4.74	4.46	0.97
CEL-HO_HEAP	3.64	4.51	4.62	4.70	4.46	4.70	4.44	0.40
CEL-HENRIKSEN	1.90	8.74	10.42	4.59	9.44	10.49	7.60	3.54
CCT-FIFO	8.02	9.67	9.73	9.71	10.06	9.71	9.48	0.73
FIXED-FIFO	4.66	5.73	5.79	5.77	5.97	5.77	5.61	0.47
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.43	12.27	12.47	13.59	12.28	21.94	13.16	4.99
CEL-SPLAY	8.25	17.80	17.08	13.01	16.63	16.12	14.82	3.62
CEL-HO_HEAP	10.91	18.62	18.94	19.27	18.30	19.24	17.55	3.27
CEL-HENRIKSEN	7.06	24.59	30.39	14.42	27.46	31.40	22.55	9.74
CCT-FIFO	24.81	47.88	47.95	47.91	57.00	47.92	45.58	10.80
FIXED-FIFO	14.44	27.58	27.72	27.67	32.68	27.65	26.29	6.14
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	7.96	4.00	3.99	3.98	3.33	3.99	4.54	1.70
FIXED-FIFO	7.88	3.95	3.94	3.94	3.29	3.94	4.49	1.68
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	235.09	243.79	252.23	266.99	240.94	624.69	310.62	154.26
CEL-SPLAY	248.65	428.77	421.69	360.86	413.28	413.21	381.08	69.19
CEL-HO_HEAP	385.55	609.43	612.12	618.02	602.74	616.54	574.07	92.52
CEL-HENRIKSEN	398.37	576.33	659.13	462.38	635.41	689.33	570.16	116.23
CCT-FIFO	513.15	883.03	879.05	879.31	1027.40	880.19	843.69	172.28
FIXED-FIFO	476.21	809.35	805.90	806.09	938.98	806.26	773.80	155.07
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.67	0.40	0.42	0.38	0.41	0.22	0.41	0.14
CEL-SPLAY	0.63	0.30	0.32	0.41	0.32	0.33	0.39	0.13
CEL-HO_HEAP	0.46	0.29	0.30	0.29	0.30	0.28	0.32	0.07
CEL-HENRIKSEN	0.62	0.21	0.18	0.34	0.19	0.17	0.29	0.18
CCT-FIFO	0.29	0.16	0.17	0.17	0.14	0.16	0.18	0.06
FIXED-FIFO	0.39	0.22	0.22	0.22	0.19	0.22	0.25	0.07

Table A.1: Results for D4_R32_L1 Distribution experiment

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.62	3.50	4.56	4.73	3.85	4.72	4.16	0.57
CEL-SPLAY	3.65	3.57	4.61	4.73	3.90	4.73	4.20	0.55
CEL-HO_HEAP	3.63	3.49	4.53	4.70	3.82	4.67	4.14	0.55
CEL-HENRIKSEN	3.64	3.59	4.64	4.72	3.91	4.76	4.21	0.56
CCT-FIFO	3.17	2.96	3.75	3.73	3.31	3.74	3.44	0.34
FIXED-FIFO	3.00	2.99	3.85	4.08	3.31	3.96	3.53	0.49
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.63	2.86	3.04	3.27	2.70	2.83	2.72	0.57
CEL-SPLAY	2.09	2.58	2.75	3.18	2.19	2.89	2.62	0.42
CEL-HO_HEAP	1.21	1.41	1.49	1.58	1.33	1.55	1.43	0.14
CEL-HENRIKSEN	1.81	2.74	2.85	3.27	2.12	2.92	2.62	0.54
CCT-FIFO	3.45	3.95	3.96	3.96	4.06	3.96	3.89	0.22
FIXED-FIFO	5.52	6.65	6.69	6.72	6.88	6.71	6.53	0.50
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.24	9.61	10.12	10.91	9.11	10.48	9.41	1.68
CEL-SPLAY	6.92	10.47	10.90	11.23	9.29	11.25	10.01	1.68
CEL-HO_HEAP	5.73	8.30	8.69	9.17	7.83	9.04	8.13	1.27
CEL-HENRIKSEN	6.64	10.27	10.71	10.91	8.87	11.05	9.74	1.71
CCT-FIFO	16.00	29.64	29.72	29.77	34.61	29.79	28.25	6.31
FIXED-FIFO	17.52	32.88	33.17	33.59	38.18	33.38	31.45	7.11
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	0.93	0.46	0.46	0.46	0.38	0.46	0.53	0.20
FIXED-FIFO	0.49	0.25	0.25	0.25	0.21	0.25	0.28	0.10
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	238.09	246.90	259.09	268.31	243.62	351.40	267.90	42.36
CEL-SPLAY	248.85	377.68	371.20	312.58	364.63	363.00	339.66	50.17
CEL-HO_HEAP	316.23	470.95	473.89	479.69	465.02	478.23	447.34	64.45
CEL-HENRIKSEN	338.49	431.13	443.06	341.22	433.26	447.16	405.72	51.37
CCT-FIFO	740.59	1351.86	1350.41	1350.40	1595.69	1351.27	1290.04	286.42
FIXED-FIFO	446.83	763.47	762.61	762.32	888.43	762.69	731.06	148.04
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.71	0.49	0.50	0.47	0.53	0.46	0.53	0.09
CEL-SPLAY	0.74	0.45	0.48	0.50	0.52	0.47	0.53	0.11
CEL-HO_HEAP	0.78	0.53	0.56	0.54	0.58	0.52	0.59	0.10
CEL-HENRIKSEN	0.70	0.44	0.45	0.51	0.50	0.44	0.51	0.10
CCT-FIFO	0.35	0.19	0.20	0.20	0.17	0.20	0.22	0.06
FIXED-FIFO	0.34	0.19	0.19	0.19	0.17	0.19	0.21	0.06

Table A.2: Results for D0.25_R1_L1 Distribution experiment

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.71	4.00	5.07	5.04	4.64	5.20	4.61	0.62
CEL-SPLAY	3.72	4.10	5.17	5.07	4.73	5.15	4.66	0.61
CEL-HO_HEAP	3.84	4.11	5.19	5.15	4.75	5.18	4.70	0.59
CEL-HENRIKSEN	3.72	4.07	5.22	5.10	4.76	5.21	4.68	0.64
CCT-FIFO	0.60	2.37	2.98	2.84	3.10	2.98	2.48	0.95
FIXED-FIFO	0.97	2.71	3.41	3.26	3.38	3.41	2.86	0.96
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.44	4.14	4.26	4.46	4.14	5.51	3.99	1.35
CEL-SPLAY	2.84	5.73	5.59	3.99	4.93	5.25	4.72	1.11
CEL-HO_HEAP	3.81	4.52	4.62	4.73	4.26	4.69	4.44	0.35
CEL-HENRIKSEN	1.87	6.48	9.09	5.13	7.11	9.02	6.45	2.71
CCT-FIFO	1.09	3.45	3.49	3.45	3.74	3.50	3.12	1.00
FIXED-FIFO	1.28	5.64	5.71	5.62	6.23	5.72	5.04	1.85
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.13	12.17	12.44	12.70	12.04	21.22	12.78	4.83
CEL-SPLAY	8.02	18.77	18.25	14.07	16.48	17.36	15.49	4.02
CEL-HO_HEAP	11.14	18.31	18.56	18.99	17.26	18.82	17.18	3.02
CEL-HENRIKSEN	6.84	17.75	25.45	15.36	19.75	25.55	18.45	7.01
CCT-FIFO	1.86	11.93	11.97	11.73	14.89	12.00	10.73	4.51
FIXED-FIFO	2.37	14.74	14.78	14.44	18.07	14.82	13.20	5.48
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	15.66	1.75	1.74	1.74	1.29	1.74	3.99	5.72
FIXED-FIFO	7.92	0.88	0.88	0.88	0.65	0.88	2.02	2.89
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	311.73	254.24	274.02	250.97	247.41	936.66	379.17	274.15
CEL-SPLAY	248.37	437.97	431.30	394.48	418.10	423.61	392.30	72.07
CEL-HO_HEAP	384.62	596.40	598.16	604.13	582.73	601.65	561.28	86.87
CEL-HENRIKSEN	399.98	471.97	589.17	479.09	515.49	599.67	509.23	75.95
CCT-FIFO	182.99	484.68	481.99	478.33	595.63	480.95	450.76	138.91
FIXED-FIFO	164.31	332.37	329.33	325.06	391.18	328.55	311.80	76.47
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.67	0.41	0.42	0.42	0.43	0.22	0.43	0.14
CEL-SPLAY	0.66	0.29	0.31	0.39	0.34	0.32	0.39	0.14
CEL-HO_HEAP	0.48	0.30	0.31	0.30	0.32	0.30	0.34	0.07
CEL-HENRIKSEN	0.65	0.30	0.22	0.35	0.28	0.22	0.34	0.16
CCT-FIFO	1.29	0.40	0.42	0.43	0.35	0.42	0.55	0.36
FIXED-FIFO	1.26	0.38	0.40	0.40	0.33	0.39	0.53	0.36

Table A.3: Results for D4_R1_L0.125 Distribution experiment

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.70	3.97	5.05	5.01	4.62	5.19	4.59	0.62
CEL-SPLAY	3.71	4.07	5.15	5.00	4.72	5.13	4.63	0.60
CEL-HO_HEAP	3.83	4.09	5.19	5.13	4.75	5.17	4.69	0.60
CEL-HENRIKSEN	3.71	4.09	5.23	5.04	4.79	5.22	4.68	0.64
CCT-FIFO	0.62	0.85	1.06	1.05	1.14	1.06	0.96	0.19
FIXED-FIFO	0.97	1.22	1.53	1.53	1.58	1.53	1.39	0.25
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.59	4.06	4.22	4.72	4.13	7.03	4.29	1.73
CEL-SPLAY	2.84	5.54	5.30	3.80	5.20	4.93	4.60	1.06
CEL-HO_HEAP	3.80	4.76	4.87	4.96	4.72	4.95	4.68	0.44
CEL-HENRIKSEN	1.86	8.75	10.08	4.51	9.39	10.33	7.49	3.48
CCT-FIFO	1.10	1.68	1.71	1.71	1.87	1.71	1.63	0.27
FIXED-FIFO	1.29	2.36	2.41	2.43	2.72	2.42	2.27	0.50
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.16	11.96	12.20	13.31	11.97	21.48	12.85	4.93
CEL-SPLAY	8.01	18.11	17.37	13.06	16.93	16.39	14.98	3.84
CEL-HO_HEAP	11.12	19.21	19.53	19.89	18.89	19.84	18.08	3.43
CEL-HENRIKSEN	6.83	24.53	29.48	14.08	27.25	30.97	22.19	9.62
CCT-FIFO	1.89	3.48	3.48	3.48	4.04	3.48	3.31	0.73
FIXED-FIFO	2.38	4.58	4.60	4.62	5.28	4.61	4.34	1.00
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	14.88	7.48	7.47	7.46	6.23	7.46	8.50	3.17
FIXED-FIFO	7.88	3.95	3.94	3.94	3.29	3.94	4.49	1.68
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	235.09	243.79	252.22	266.97	240.91	624.49	310.58	154.18
CEL-SPLAY	248.65	428.79	421.73	360.61	413.25	413.23	381.04	69.22
CEL-HO_HEAP	385.55	609.42	612.10	618.02	602.75	616.53	574.06	92.51
CEL-HENRIKSEN	398.37	578.22	657.33	462.16	634.25	690.12	570.08	116.05
CCT-FIFO	185.28	233.75	230.23	227.87	247.92	229.46	225.75	21.13
FIXED-FIFO	165.19	193.97	190.29	188.16	200.23	189.41	187.87	11.94
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.70	0.42	0.43	0.39	0.43	0.24	0.44	0.15
CEL-SPLAY	0.66	0.30	0.33	0.42	0.33	0.34	0.40	0.14
CEL-HO_HEAP	0.48	0.29	0.30	0.29	0.30	0.29	0.33	0.08
CEL-HENRIKSEN	0.65	0.22	0.19	0.36	0.20	0.18	0.30	0.18
CCT-FIFO	1.29	0.83	0.93	0.93	0.84	0.90	0.95	0.17
FIXED-FIFO	1.26	0.79	0.87	0.87	0.79	0.84	0.90	0.18

Table A.4: Results for D4_R1_L1 Distribution experiment

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.68	3.94	5.01	4.96	4.57	5.18	4.56	0.62
CEL-SPLAY	3.70	4.05	5.14	4.98	4.69	5.12	4.61	0.61
CEL-HO_HEAP	3.82	4.07	5.17	5.12	4.73	5.16	4.68	0.60
CEL-HENRIKSEN	3.70	4.08	5.23	4.99	4.73	5.08	4.63	0.61
CCT-FIFO	0.63	0.67	0.84	0.83	0.86	0.83	0.78	0.10
FIXED-FIFO	0.96	0.97	1.22	1.19	1.22	1.21	1.13	0.13
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.62	3.92	4.19	4.78	4.02	7.85	4.40	2.01
CEL-SPLAY	2.84	5.50	5.69	4.19	4.91	5.34	4.74	1.08
CEL-HO_HEAP	3.78	4.67	4.81	4.95	4.62	4.92	4.63	0.43
CEL-HENRIKSEN	1.87	9.23	10.92	4.54	7.87	6.39	6.80	3.27
CCT-FIFO	1.11	1.41	1.44	1.44	1.55	1.44	1.40	0.15
FIXED-FIFO	1.30	1.86	1.90	1.90	2.09	1.90	1.82	0.27
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.19	11.68	12.05	13.53	11.76	21.41	12.77	4.93
CEL-SPLAY	8.00	17.83	17.97	13.66	16.15	17.07	15.11	3.83
CEL-HO_HEAP	11.11	18.89	19.32	19.85	18.54	19.73	17.91	3.37
CEL-HENRIKSEN	6.82	26.22	32.62	13.94	22.16	17.34	19.85	9.16
CCT-FIFO	1.93	2.74	2.75	2.75	3.05	2.75	2.66	0.38
FIXED-FIFO	2.39	3.51	3.52	3.51	3.91	3.52	3.40	0.52
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	14.18	9.50	9.49	9.47	8.37	9.48	10.08	2.06
FIXED-FIFO	7.82	5.24	5.23	5.22	4.62	5.23	5.56	1.13
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	231.28	239.36	241.17	274.98	237.94	491.77	286.08	101.93
CEL-SPLAY	248.37	422.19	410.72	352.44	407.31	404.37	374.24	66.26
CEL-HO_HEAP	386.59	605.92	609.76	618.52	599.46	615.17	572.57	91.36
CEL-HENRIKSEN	397.50	605.11	697.34	458.01	558.14	466.35	530.41	110.66
CCT-FIFO	186.14	212.93	210.26	211.05	220.72	210.49	208.60	11.68
FIXED-FIFO	164.76	180.64	177.84	179.41	184.43	178.03	177.52	6.70
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.70	0.42	0.44	0.38	0.44	0.25	0.44	0.15
CEL-SPLAY	0.66	0.31	0.32	0.41	0.34	0.33	0.39	0.14
CEL-HO_HEAP	0.48	0.29	0.30	0.29	0.31	0.29	0.33	0.08
CEL-HENRIKSEN	0.65	0.21	0.18	0.40	0.24	0.33	0.33	0.17
CCT-FIFO	1.29	0.92	1.04	1.04	0.96	1.00	1.04	0.13
FIXED-FIFO	1.26	0.89	1.00	0.99	0.92	0.96	1.00	0.13

Table A.5: Results for D4_R1_L2 Distribution experiment

Algorithm	Distribution						Mean	Standard Deviation
	Constant	Exponential	Uniform	Biased	Bi-model	Triangular		
Model Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	3.70	3.99	5.06	5.01	4.65	5.27	4.61	0.64
CEL-SPLAY	3.71	4.12	5.20	5.05	4.78	5.17	4.67	0.62
CEL-HO_HEAP	3.87	4.15	5.24	5.19	4.82	5.22	4.75	0.60
CEL-HENRIKSEN	3.70	4.13	5.29	5.10	4.83	5.27	4.72	0.66
CCT-FIFO	0.08	0.11	0.14	0.14	0.16	0.14	0.13	0.03
FIXED-FIFO	0.13	0.19	0.24	0.24	0.27	0.24	0.22	0.05
Kernel Level Cache Behavior Cache Misses (%)								
CEL-CALENDAR	1.60	4.20	4.30	4.95	4.39	12.74	5.36	3.80
CEL-SPLAY	2.92	7.41	7.15	5.64	7.14	6.74	6.17	1.71
CEL-HO_HEAP	5.84	7.30	7.42	7.49	7.29	7.49	7.14	0.64
CEL-HENRIKSEN	2.63	13.19	15.20	9.46	14.55	15.57	11.77	5.00
CCT-FIFO	0.60	0.84	0.86	0.86	0.90	0.85	0.82	0.11
FIXED-FIFO	0.61	1.07	1.10	1.11	1.08	1.09	1.01	0.19
Amortized Aggregate Cache Behavior Cache Misses / Event								
CEL-CALENDAR	6.17	12.20	12.30	13.73	12.40	55.93	18.79	18.39
CEL-SPLAY	8.14	23.37	22.46	17.48	22.14	21.29	19.15	5.77
CEL-HO_HEAP	16.28	29.52	29.87	30.29	29.18	30.23	27.56	5.54
CEL-HENRIKSEN	8.25	40.24	53.61	26.70	48.60	58.14	39.26	18.81
CCT-FIFO	0.68	1.12	1.12	1.12	1.18	1.12	1.05	0.19
FIXED-FIFO	0.74	1.47	1.48	1.48	1.49	1.48	1.36	0.30
Events Per LP Execution Events / LP Execution								
CEL-CALENDAR	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-SPLAY	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HO_HEAP	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CEL-HENRIKSEN	2.00	1.00	1.00	1.00	1.00	1.00	1.17	0.41
CCT-FIFO	119.22	59.82	59.70	59.74	49.81	59.64	67.99	25.41
FIXED-FIFO	63.00	31.61	31.56	31.53	26.30	31.54	35.92	13.43
Amortized Kernel Level Computation Cost Instructions / Event								
CEL-CALENDAR	232.61	241.56	246.24	268.18	241.17	1559.80	464.93	536.51
CEL-SPLAY	248.57	464.27	457.13	395.96	447.29	448.62	410.30	82.83
CEL-HO_HEAP	437.49	713.31	716.00	721.92	706.42	720.43	669.26	113.68
CEL-HENRIKSEN	443.03	772.31	971.90	615.25	897.70	1051.20	791.90	229.84
CCT-FIFO	154.28	215.29	213.15	207.97	200.77	214.85	201.05	23.56
FIXED-FIFO	151.76	209.97	207.80	202.92	194.38	209.50	196.05	22.47
Event Rate (*10^6 events / second)								
CEL-CALENDAR	0.69	0.40	0.42	0.39	0.41	0.09	0.40	0.19
CEL-SPLAY	0.64	0.24	0.25	0.31	0.25	0.26	0.33	0.16
CEL-HO_HEAP	0.35	0.20	0.20	0.20	0.21	0.20	0.23	0.06
CEL-HENRIKSEN	0.61	0.15	0.11	0.22	0.13	0.10	0.22	0.20
CCT-FIFO	1.67	1.08	1.25	1.28	1.21	1.19	1.28	0.20
FIXED-FIFO	1.68	1.05	1.21	1.23	1.19	1.16	1.26	0.22

Table A.6: Results for D32_R1_L1 Distribution experiment

Bibliography

[1] BLANCHARD, T. D., LAKE, T. W., AND TURNER, S. J. Cooperative acceleration: Robust conservative distributed discrete event simulation. In *PADS '94: Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (1994), pp. 58–64.

[2] BROWN, R. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications ACM 31*, 10 (1988), 1220–1227.

[3] BRYANT, R. E. Simulation of packet communication architecture computer systems. Technical Report TR-188, MIT Labratory for Computer Science, 1977.

[4] CHANDY, K. M., AND MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering SE-5*, 5 (1979), 440–452.

[5] CHANDY, K. M., AND MISRA, J. Asynchronous distributed simulation via a sequence of parallel computations. *Communications ACM 24*, 4 (1981), 198–206.

[6] CURRY, R., KIDDLE, C., SIMMONDS, R., AND UNGER, B. Sequential performance of asynchronous conservative PDES algorithms. In *PADS '05: Proceedings of the 2005 Workshop on Principles of Advanced and Distributed Simulation* (2005), IEEE Computer Society, pp. 217–226.

[7] CURRY, R., SIMMONDS, R., AND UNGER, B. Simulating diffserv in conservative PDES. In *Proceedings of 2003 Winter Simulation Conference* (2003), pp. 658–666.

[8] DAVEY, D., AND VAUCHER, J. Self-optimizing partition sequencing sets for discrete event simulation. In *INFOR* (1980), vol. 18 of *1*, pp. 21–41.

[9] DICKENS, P. M., AND REYNOLDS, JR., P. F. SRADS with local rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990), vol. 22 of *SCS Simulation Series*, pp. 161–164.

[10] FRANTA, W., AND MALY, K. An efficient data structure for the simulation event set. *Communications. ACM 8*, 20 (1977), 596–602.

[11] FUJIMOTO, R. M. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation 6*, 3 (1989), 211–239.

[12] FUJIMOTO, R. M. Parallel discrete event simulation. *Communications ACM 33*, 10 (1990), 30–53.

[13] FUJIMOTO, R. M. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990), vol. 22, pp. 23–28.

[14] FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems.* John Wiley & Sons, Inc., New York, 2000.

[15] GONNET, G. H. Heaps applied to event driven mechanisms. *Communications ACM 19*, 7 (1976), 417–418.

[16] GORDON, G. The development of the general purpose simulation system (gpss). In *The first ACM SIGPLAN conference on History of programming languages* (1978), ACM Press, pp. 183–198.

[17] HENRIKSEN, J. O. An improved events list algorithm. In *Proceedings of the 9th Winter Simulation Conference* (1977), pp. 546–557.

[18] HENRIKSEN, J. O. Event list management - a tutorial. In *Proceedings of the 15th Winter Simulation Conference* (1983), IEEE Press, pp. 543–551.

[19] JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (1985), 404–425.

[20] JONES, D. W. An empirical comparison of priority-queue and event-set implementations. *Communications ACM 29*, 4 (1986), 300–311.

[21] KIDDLE, C., SIMMONDS, R., AND UNGER, B. Performance of a mixed shared/distributed memory parallel network simulator. In *PADS '04: Proceedings of the 18th Workshop on Parallel and Distributed Simulation* (2004), pp. 17–25.

[22] KINGSTON, J. H. The amortized complexity of Henriksen's algorithm. Technical Report 85-06, University of Iowa, 1985.

[23] LIU, J., NICOL, D. M., AND TAN, K. Lock-free scheduling of logical processes in parallel simulation. In *PADS '01: Proceedings of the 15th Workshop on Parallel and Distributed Simulation* (2001), pp. 22–31.

[24] LUBACHEVSKY, B. D. Bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1988), pp. 183–191.

[25] LUBACHEVSKY, B. D. Efficient distributed event-driven simulations of multiple-loop networks. *Communications ACM 32*, 1 (1989), 111–123.

[26] MARÍN, M. On the pending event set and binary tournaments. In *10th Annual SCS European Simulation Symposium* (1998), Society for Computer Simulation European Publishing House, pp. 110–114.

[27] McCormack, W. M., and Sargent, R. G. Analysis of future event set algorithms for discrete event simulation. *Communications ACM 24*, 12 (1981), 801–812.

[28] Nicol, D. M. Principles of conservative parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference* (1996), pp. 128–135.

[29] Porter, T., and Simon, I. Random insertion into a priority queue structure. *IEEE Transactions on Software Engineering 1*, 3 (1975), 292–298.

[30] Reynolds, Jr., P. F. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference* (1988), pp. 325–332.

[31] Rönngren, R., Riboe, J., and Ayani, R. A comparative study of some priority queues suitable for implementation of the pending event set. *ACM Transactions on Modeling and Computer Simulation 7* (1993), 157–209.

[32] Rönngren, R., Riboe, J., and Ayani, R. Lazy queue: A new approach to implementation of the pending-event set. *Internation Journal of Computer Simulation 3* (1993), 303–332.

[33] Simmonds, R., Bradford, R., and Unger, B. Applying parallel discrete event simulation to network emulation. In *PADS '00: Proceedings of the 14th Workshop on Parallel and Distributed Simulation* (2000), pp. 15–22.

[34] Simmonds, R., Kiddle, C., and Unger, B. Addressing blocking and scalability in critical channel traversing. In *PADS '02: Proceedings of the 16th Workshop on Parallel and Distributed Simulation* (2002), pp. 17–24.

[35] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *Journal ACM 32*, 3 (1985), 652–686.

[36] STEINMAN, J. S. Breathing time warp. In *PADS '93: Proceedings of the seventh workshop on Parallel and Distributed Simulation* (1993), ACM Press, pp. 109–118.

[37] STEINMAN, J. S. Discrete-event simulation and the event horizon. In *PADS '94: Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (1994), pp. 39–49.

[38] UNGER, B., XIAO, Z., CLEARY, J., TSAI, J.-J., AND WILLIAMSON, C. Parallel shared-memory simulator performance for large atm networks. *ACM Transactions on Modeling and Computer Simulation 10*, 4 (2000), 358–391.

[39] VAUCHER, J. G., AND DUVAL, P. A comparison of simulation event list algorithms. *Communications ACM*, 4 (1975), 223–230.

[40] WOOD, K. R., AND TURNER, S. J. A generalized carrier-null method for conservative parallel simulation. In *PADS '94: Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (1994), pp. 50–57.

[41] XIAO, Z., SIMMONDS, R., UNGER, B., AND CLEARY, J. Fast cell level ATM network simulation. In *Proceedings of the 2002 Winter Simulation Conference* (2002), pp. 712–719.

[42] XIAO, Z., UNGER, B., SIMMONDS, R., AND CLEARY, J. Scheduling critical channels in conservative parallel discrete event simulation. In *PADS '99: Proceedings of the 13th workshop on Parallel and distributed simulation* (1999), IEEE Computer Society, pp. 20–28.