Event Pool Structures for PDES on Many-Core Beowulf Clusters

Tom Dickman
School of Electronic and Computing Systems
Cincinnati, OH 45221-0030
tdickman@gmail.com

Sounak Gupta
School of Electronic and Computing Systems
Cincinnati, OH 45221-0030
sounak.besu@gmail.com

Philip A. Wilsey
School of Electronic and Computing Systems
Cincinnati, OH 45221-0030
wilseypa@gmail.com

ABSTRACT

Multi-core and many-core processing chips are becoming widespread and are now being widely integrated into Beowulf clusters. This poses a challenging problem for distributed simulation as it now becomes necessary to extend the algorithms to operate on a platform that includes both shared memory and distributed memory hardware. Furthermore, as the number of on-chip cores grows, the challenges for developing solutions without significant contention for shared data structures grows. This is especially true for the pending event list data structures where multiple execution threads attempt to schedule the next event for execution. This problem is especially aggravated in parallel simulation, where event executions are generally fine-grained leading quickly to non-trivial contention for the pending event list.

This manuscript explores the design of the software architecture and several data structures to manage the pending event sets for execution in a Time Warp synchronized parallel simulation engine. The experiments are especially targeting multi-core and many-core Beowulf clusters containing 8-core to 48-core processors. These studies include a two-level structure for holding the pending event sets using three different data structures, namely: splay trees, the STL multiset, and ladder queues. Performance comparisons of the three data structures using two architectures for the pending event sets are presented.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming, distributed programming*

; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*parallel, distributed, discrete event*

General Terms

Algorithms, Performance

SIGSIM-PADS'13, May 19–22, 2013, Montréal, Québec, Canada.

Keywords

Time Warp, pending event lists, multi-core, threads, Beowulf clusters

1. INTRODUCTION

Trends in desktop microprocessors have shown that the shift from multi-core to many-core is on the horizon. The road maps of all the major processor providers (Intel, AMD, and IBM) clearly show this progression. Both Intel and AMD have commodity multi-core processors with hardware support for up to 16 simultaneous threads. These can be configured in multi-chip motherboards for a hardware SMP configuration supporting as many as 64 simultaneous threads. IBM's Power7 product is expected to support up to 32 threads per chip [9]. Intel has demonstrated a 48 core Single-Chip Cloud computer (SCC) [8] and a 50-core Knights Corner (both implementing x86 compatible cores). Following these patterns, it is clear that desktop processors may soon contain hardware support providing capabilities for hundreds of simultaneously executing threads.

As the number of cores increases, it is increasingly challenging for application software to take advantage of the additional parallel processing capabilities. The key to successfully harnessing this power lies with new advances to better introduce parallelism into the practice of computer system and software development [6]. From the standpoint of parallel discrete event driven simulation (PDES), one of the key challenge areas lies in the need for solutions providing effective, contention-free pending event list management. This is especially true since many discrete event simulation models tend to have relatively fine-grained computational requirements making frequent access to the pending event set necessary.

In this manuscript, we examine the design of a software architecture and data structures to hold the pending event sets on each node of a many-core Beowulf cluster executing Time Warp synchronized parallel simulations. In particular, we examine a two-level structure to manage the pending event sets and evaluate three different data structures (splay tree, STL multiset, and ladder queues) within the two-level structure. The two-level structure explores a multiple-worker-thread/multiple-event-pool model that is ultimately designed for many-core processors with lightweight on-chip load balancing. In this paper the preliminary implementation using various configurations of the event-pool model without dynamic load balancing is examined. Explorations with the ladder queue are especially significant as we have future plans to exploit its bucketing of events to build a

shared event pool that is fully managable with atomic move instructions — implementing a wait-free pending event list. This concept is more fully discussed in Section 6.

The remainder of this manuscript is organized as follows. Section 2 reviews the previous literature on pending event list management in Time Warp simulation. Section 3 provides a brief background review of the ladder queue data structure. Section 4 reviews the software architecture of the Time Warp simulation kernel (WARPED) that is used in this work. Section 5 presents the results of our experimental analysis. Section 6 contains a discussion on some possible extensions of the event pool structures studied in this work. Finally, Section 7 presents some concluding remarks.

2. RELATED WORK

Events in WARPED are organized into two event pools, namely: `Unprocessed` and `Processed` [18]. The incoming events, waiting to be executed, are stored in the `Unprocessed` pool. To a certain degree, the WARPED design inherently incorporates tolerance towards the execution order of events in their causal order. The existing `Unprocessed` events are optimistically executed and since the processed events may be required for later reexecution, they are moved to a holding list called the `Processed` queue [12]. This model is used elsewhere in other time warp synchronized simulators. For example, Ronngren *et al* [18] discuss the use of `Linear List` which is essentially a doubly linked list structure that can hold all events (processed and unprocessed) along with information about the execution status of each event. This design allows for simple implementation, quick and efficient rollback as well as efficient `fossil` collection. The effectiveness of this structure is questioned in [18] because of inefficient insertion into a significantly large `Unprocessed` event pool. They suggest the use of an improved skew heap as a possible data structure for storing the events to be executed.

Prasad *et al* [14] explored parallelized Calendar Queues [3] for medium to coarse-grained optimistic simulators. Separate calender queues are allocated to each processor. They observed from a comparative study between the global-queue-based and local-queue-based simulators that both achieved excellent load balancing with the former being faster with fewer rollbacks. They also exploited the property of *grain packing* in parallel heap for fine-grained simulation.

Santoro *et al* [20] explored a version of a Least-TimeStamp-First (LTSF) Scheduler that is somewhat similar to the Calendar Queue, except it is created using an array and a hierarchical bitmap. The main advantage that this data structure provides versus the Calendar Queue is that access to it is in constant-time, and it has low-overhead.

3. LADDER QUEUE

The ladder queue data structure [22] is a bucket based priority queue that shares many characteristics with calendar queues [3]. The chief advantage of a ladder queue is that the buckets (months) storing events are dynamically split when the number of stored events exceed some threshold. Thus, instead of resizing the entire data structure as is done in calendar queues, ladder queues simply break the bucket in question up into another collection of buckets. An illustration of the principle components of a ladder queue is shown in Figure 1. A brief description of the operation of a ladder queue is presented below.

Initially the ladder is empty and incoming events are linked (unsorted) together in the `Top` component. As they arrive, the minimum and maximum timestamps on the events in `Top` are recorded. When the first "dequeue event" operation occurs, all of the events in `Top` are transferred to the buckets in `Rung[1]`. The number of buckets in `Rung[1]` is a dynamically configurable parameter. Each bucket in `Rung[1]` is defined to hold events for a time range that equally subdivides the range between the minimum and maximum timestamps of all events that were originally in `Top`. The events from `Top` are placed (unsorted) into the bucket in `Rung[1]` corresponding to its timestamp.

The buckets in the ladder queue are defined to hold a maximum number of events. Whenever the number of events in a bucket exceed this maximum, a new lower `Rung` in the ladder is defined and those events are redistributed into it, and so on. An example of this redistribution is shown in Figure 1. Note the redistribution of the events from the 5th bucket in `Rung[1]` into `Rung[2]`).

To complete the dequeue event, all events from the leftmost non-empty time bucket (containing events with the smallest timestamps) are sorted and placed in `Bottom`. The dequeue operation then pulls the first event out of `Bottom`. Successive dequeue event operations pull from `Bottom` until it is empty which then triggers another pull of events from the leftmost non-empty bucket in the ladder. Once all of the buckets in the rungs of the ladder are empty, events are once again pulled down into the rungs of the ladder from `Top`.

After the initial ladder is constructed, incoming events are distributed into the ladder according to their timestamp. Those falling into the time window for the rungs in the ladder are placed into the corresponding rung/bucket location (including being sorted into `Bottom` if that is where it falls). Incoming events with a timestamp greater than the maximum timestamp used to create the ladder rungs are placed in `Top` (where new minimum/maximum timestamps are recorded).

Conceptually, the ladder queue is organized into epochs where the `Bottom` and `Rung` elements hold events with timestamps between t and $t + \Delta t$ while the `Top` element holds events with timestamps above $t + \Delta t$. Incoming events populate the ladder queue elements accordingly and once the dequeue operations empty the `Bottom` and `Rung` elements, another ladder queue epoch occurs and the events in `Top` repopulate the ladder `Rungs` and `Bottom`. Coincident with a new epoch, the time range of the ladder (t to $t + \Delta t$) is redefined by the minimum and maximum timestamps of events pulled from `Top`. There are a few additional special case situations that are more fully described in [22].

4. WARPED: A TIME WARP SIMULATION KERNEL

WARPED is a discrete event simulation kernel that implements the Time Warp synchronization protocol [10]. It was originally designed and optimized for executing parallel simulations on a Beowulf Cluster containing single core processors. It is highly configurable and incorporates many different sub-algorithms (*e.g.,* periodic checkpointing [5], and lazy, aggressive, and dynamic cancellation [16]) of the Time

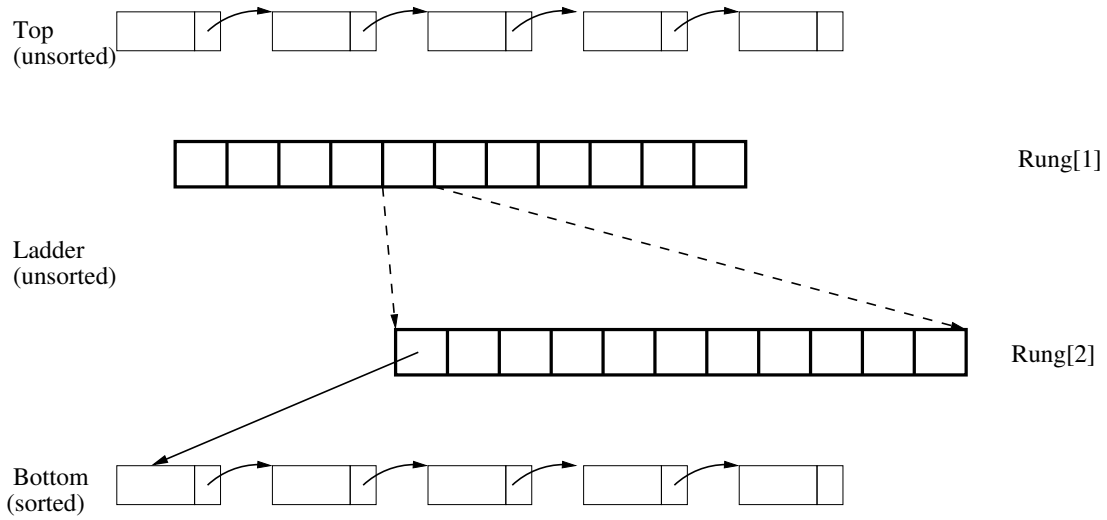

Figure 1: Illustration of the Ladder Queue Structure

Warp mechanism [5]. Structurally, the Logical Processes (LPs) of a simulation are grouped together on each processing node where the LPs are scheduled according to a Least-Timestamp-First (LTSF) event scheduling policy. The node architecture reduces the Time Warp housekeeping functions such as GVT estimation, termination detection, and fossil collection into a set of common services for the entire population of LPs on that node. This architecture is similar to that reported in [1] and [15].

Most recently several attempts to build a threaded extension of WARPED have been pursued [11, 12]. These studies have produced a solution that works reasonably well for smaller multi-core processors. The overall design structure depicting the main pending event pool and the executing threads is shown in Figure 2. A threaded instance of WARPED contains a manager thread and one or more worker threads. The *manager thread* (labeled `M` in Figure 2) processes the Time Warp housekeeping functions and also processes the receipt and transmission of event messages exchanged with remote nodes in the cluster (local event insertion is performed by the worker threads). Additional details on the operation of the manager thread are available in [12]. The *worker threads* (depicted as `W0` and `Wn` in Figure 2) are responsible for dequeueing and executing pending events and generating new events accordingly. The pending event sets are organized into a two level structure as described below.

The pending event lists for each LP are maintained as independent sorted[1] lists that are independently locked. The lowest timestamped event from each LP event list is placed in a common LTSF pending event queue. The (locked) LTSF queue is sorted and used by the *worker threads* to schedule the next event for execution. After dequeuing and processing an event from the LTSF, each worker thread will then access the pending event set of the LP corresponding to the event just executed and remove the next least-timestamped event for insertion back into the LTSF queue. An abstract representation of the general event processing algorithm performed by the worker threads is shown in Figure 3.

While the above described design works well when the system is configured with only a few worker threads, once the number of worker threads exceeds 5-6, contention for the LTSF queue begins to negatively impact performance. Since the LP event pools are independently locked and since only one worker thread and the manager thread will simultaneously access the same LP event pool, contention to these structures is minimized. The principle point of contention for pending events in this architecture are at the LTSF queue. Thus, this study examines alternate designs for organizing the pending event list and especially the LTSF queue.

5. EXPERIMENTAL ANALYSIS

This study pursues two main issues with the pending event lists in the above described implementation of threaded WARPED. The first part of this study is to address the contention issue to the LTSF queue. We address this problem by creating multiple instances of the LTSF queue and partitioning the worker threads and the LPs to one of the LTSF queues. The second part of this study is the use of alternate data structures underlying the implementation of the LTSF queue. In particular, we study replacing the multiset from the C++ standard template library (STL) with two alternates, namely: (i) splay trees [21] and (ii) ladder queues [22]. These experiments are described in details below.

The following experiments are all performed on the same machine to ease comparison between different simulation configurations, and to keep results consistent. All simulations were performed on a machine with 48 cores with different configurations for the number of executing threads. The machine has four 12-core AMD Opteron 6168 processors, with each core running at a clock rate of 1.9GHz. The machine is configured with 64 GB of RAM.

The following simulation models are used for the experimental analysis:

RAID-5: This simulation model represents a Level 5 RAID (Redundant Array of Inexpensive Disks) setup as seen in

[1]Although the prospect of using a partially sorted data structure such as calendar queues [3], lazy queues [19], or ladder queues [22] is possible.

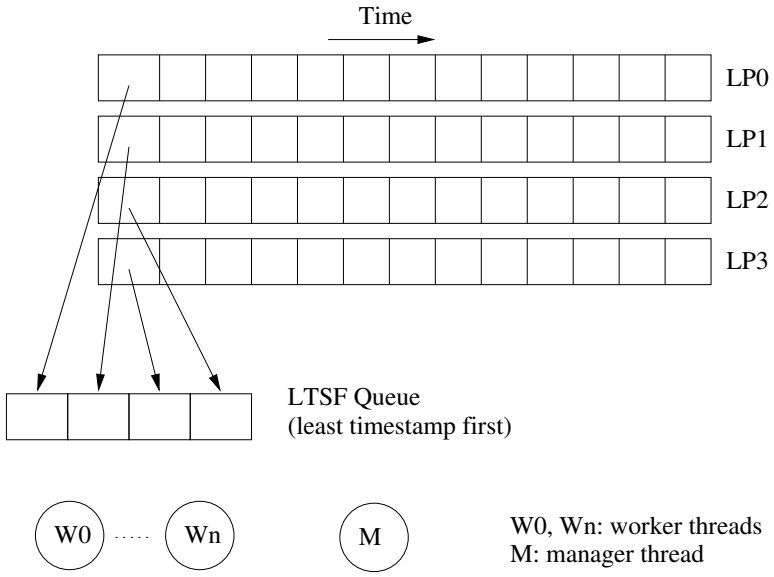

Figure 2: The principle input queues in warped.

```
worker_thread()

  lock LTSF queue
  dequeue smallest event from LTSF
  unlock LTSF queue

  while !done loop

    process event (assume from LPi)

    lock LPi queue

    dequeue smallest event from LPi

    lock LTSF queue

    insert event from LPi
    dequeue smallest event from LTSF

    unlock LTSF queue
    unlock LPi queue
  end loop
```

Figure 3: Generalized event execution loop for the worker threads. Many details have been omitted for the sake of clarity.

Figure 4. The simulation model is composed of 136 logical processes (LPs) that simulate 32 disks, 8 forks and 96 sources. The sources generate requests for data from the array, and these requests pass through the respective forks, which divide each request to the necessary disks. Each disk then responds to this request for data after a predetermined amount of time that simulates the access time to that sector of the disk. This simulation is performed until the global execution time reaches one hundred thousand seconds.

High-level ISCAS-85 benchmark circuits: The ISCAS-85 benchmarks are comprised of several different combinational logic circuits [7]. Our experiments are performed using two of these circuits, namely: C2670 and C7552. The C2670 circuit emulates a 12-bit ALU and controller. The circuit consists of an ALU with a comparator, an equality checker and several parity trees. The circuit has 157 input lines, 64 output lines, 1193 logic gates and 7 major functional blocks. These specifications translate to 1414 logical processes (LPs) for the simulation model. The second circuit, C7552, emulates a 32-bit adder/comparator. The circuit consists of a 32-bit adder, a 32-bit magnitude comparator using another 32-bit adder and a parity checker. The circuit has 207 input lines, 108 output lines, 3512 logic gates and 8 major functional blocks. These specifications translate to 3827 LPs for the simulation model.

5.1 Experiments with Multiple LTSF Queues

As previous mentioned, the shared LTSF queue in WARPED (Figure 2) starts to become a bottleneck when the number of worker threads is increased beyond approximately 5 or 6. To address this contention, we have modified the WARPED kernel to support multiple LTSF queues (Figure 5). In our preliminary implementation of multiple LTSF queues, the worker threads are divided into independent groups that are statically (and permanently) assigned to an LTSF queue (in Figure 5, the two groups of worker threads denoted W0$\cdots$Wn, represent independent sets). Likewise, the LPs are divided and statically assigned to LTSF queues. The LTSF queues

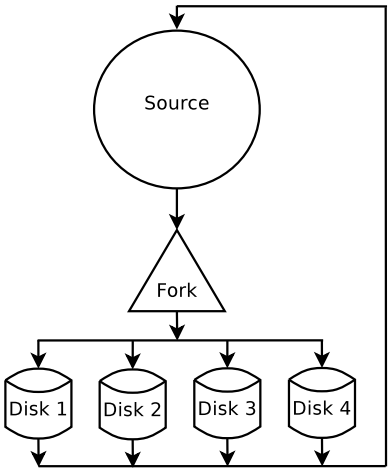

Figure 4: RAID-5 Simulation Model

are then populated with events from their assigned LPs as outlined above in the single LTSF implementation.

In our preliminary studies with multiple LTSF queues, we experienced wide variations in runtimes from run to run of the same configuration of the RAID model. Due to this instability, we started examining the partitioning of LPs to LTSF queues. As a result, several different approaches were explored for binding LPs to LTSF queues in an attempt to stabilize the system. In particular, we attempted to follow the source dependency chain of event flow between the LPs of the simulation model to guide the partitioning. However, this contributed no noticeable performance gain or stability improvement and thus, none of these results are included in this paper. For the results in this manuscript, the partitioning was a simple round robin assignment of LPs to LTSF queues. Further analysis and discussion of this performance variance issue is provided at the end of this section.

Simulations were performed with several different configurations that varied the numbers of worker threads. In particular, configurations of 4, 8, 16, 32, and 48 worker threads were studied. We also explored several different configurations of LTSF queues. Specifically we ran experiments with 1, 2, 4, and 8 LTSF queues. The number of LTSF queues was increased by a power of two so the number of threads was evenly divisible by the LTSF queues in an attempt to keep the simulations balanced. All results were obtained by taking the median of ten simulation runs. These results from these LTSF queue experiments are described below.

5.1.1 RAID-5

As shown in Figure 6, increasing the number of LTSF queues causes a decrease in execution time. In all cases, going from one LTSF queue to two queues causes a substantial decrease in the execution time. This occurs due to the decrease in contention as the number of LTSF queues increases. This decrease in simulation time continues for all thread configurations up to 4 LTSF queues. The simulation results in Figure 6 show a slight decrease in simulation time when moving from 4 to 8 LTSF queues for the 48 threaded version. This decrease can likely be attributed to the further decrease in contention when moving from 12 threads per LTSF queue to 6 threads per LTSF queue. All other simulations see an increase in simulation time when moving to 8 LTSF queues, due to the low number of threads bound to each LTSF queue.

5.1.2 ISCAS-85

As shown in Figure 7, all configurations show a decrease in simulation time when going from one to two LTSF queues. The simulations with a larger number of threads continue to show a decrease in simulation time as the number of LTSF queues is increased. All simulations have the minimum simulation time when the number of threads assigned to each LTSF queue is between four and eight. As shown in Figure 8, the c7552 simulation shows similar results, except the simulation time decreases in a more linear manner as the number of threads is increased. It also shows that the optimal number of threads per LTSF queue is between four and eight.

In previous studies with a single LTSF queue, Muthalagu [12] reports that simulation times start to increase when more than 6 threads are used in a simulation model. We arrived at a similar ratio (between four and eight threads per LTSF queue) when using the ISCAS-85 benchmark circuits as explained in the section above. By using multiple LTSF queues, we are able to reduce this contention by assigning fewer threads to each LTSF queue.

When more than eight LTSF queues are used, the simulation time is highly variant. This variance can likely be linked to the increase in the number of rollbacks that occur due to the unbalanced nature of having a large number of LTSF queues. For example, when using a single LTSF queue, the number of rollbacks was very low, almost zero. When multiple LTSF queues were used, the number of rollbacks increased significantly. An increase was expected, but this is likely due to an instability caused by the large number of LTSF queues. For example, if the LPs assigned to one LTSF queue contain fewer events, this may result in a large number of rollbacks when the LPs in one LTSF queue get too far ahead of the events in the other LTSF queue. One potential solution for this is some sort of load balancing algorithm to reassign LPs to keep any one LTSF queue from containing events with timestamps too different from the other LTSF queue.

5.2 Experiments with Ladder Queues

In this section, we present a comparative study of performance of threaded WARPED based on manipulation of the underlying data structure of LTSF queue. Prior to this study, the LTSF queues in threaded WARPED were based on the multiset object from the C++ Standard Template Library (STL). The STL multiset is an implementation of red-black tree, a type of balanced binary search tree. The asymptotic run time for insertion, deletion and lookup in a red-black tree is O(log n) [2]. We replaced the STL multiset data structure used for the LTSF queue with the Splay tree data structure. Splay tree is a self-adjusting binary search tree where recently accessed elements can be accessed again quickly. Insertion, look-up and deletion can be performed in O(log n) *amortized* [23] time. The splay tree library used in threaded WARPED is a faithful implementation of the splay tree algorithm [21].

The ladder queue data structure has already been introduced in Section 3 with a brief discussion about the potential benefits of its use as the underlying data structure for the

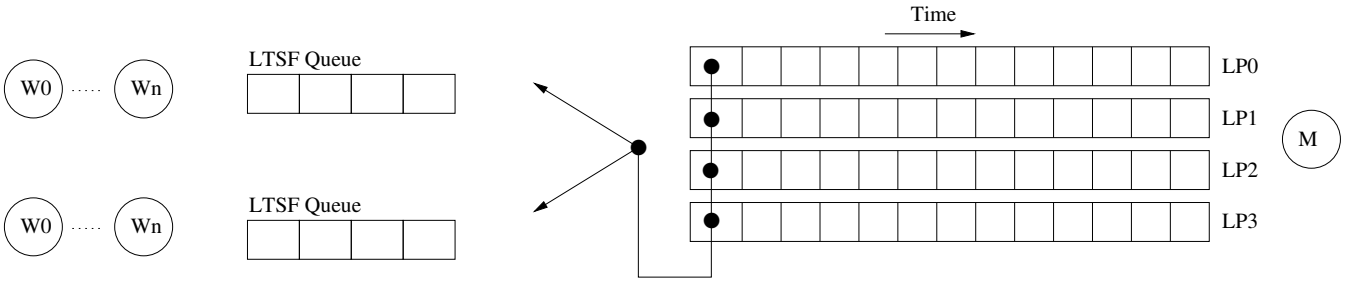

Figure 5: The principle input queues in warped.

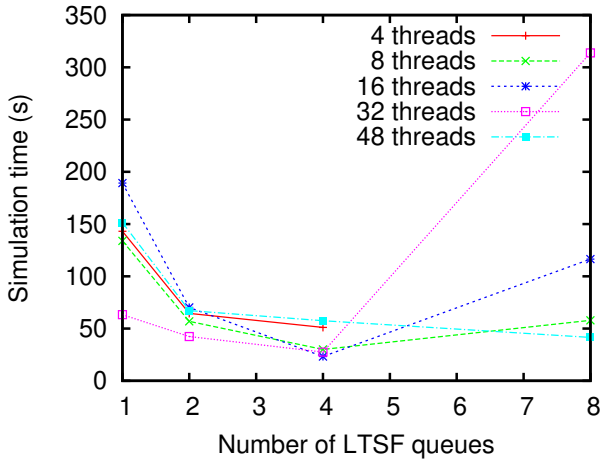

Figure 6: Performance of Multiple LTSF Queues using STL Multiset on RAID-5

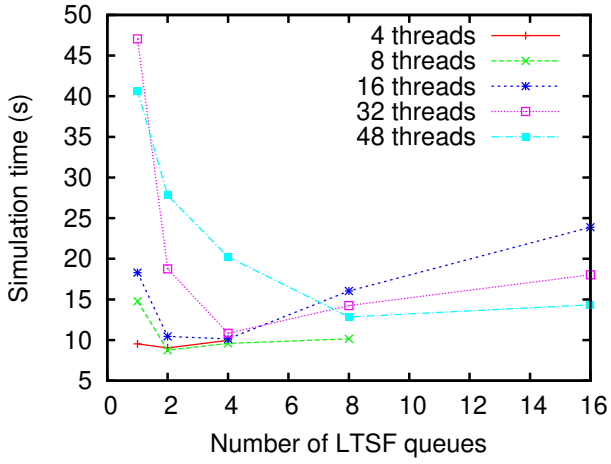

Figure 7: Performance of Multiple LTSF Queues using STL Multiset on ISCAS c2670

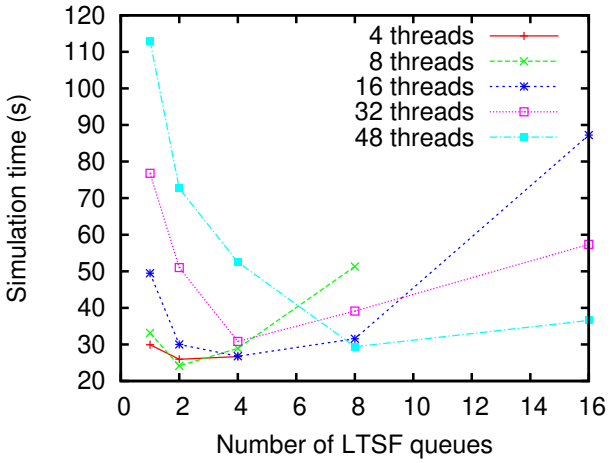

Figure 8: Performance of Multiple LTSF Queues using STL Multiset on ISCAS c7552

LTSF queues. The empirical evidence presented by Tang *et al* [22] illustrates that Ladder queue exhibits O(1) *amortized* [23] complexity.

While implementing and testing the ladder queue data structure, an issue with the original algorithm [22] arose. In particular, a problem arises when, during a redistribution event, the time window of the bucket is smaller than the number of buckets in a new lower rung. More precisely, recall that a redistribution of events in a bucket occurs when the number of events in that bucket exceeds some (configurable, but static) `threshold`. The events are redistributed into a new lower rung that has a number of buckets equal to the `threshold`. Thus, let t to $t + \Delta t$ be the size of the time window for event timestamps stored in a bucket in Rung[i]. If Δt is less than `threshold`, then a new rung cannot be defined (the time cannot be decomposed into `threshold` subparts). In this case, if the bucket in question is not the first non-empty bucket of the last rung, our algorithm simply allows the bucket size to grow beyond the `threshold` and a redistribution event does not occur.

When the number of events in the first non-empty bucket of the last rung exceeds the `threshold`, we allow the events of that bucket to be moved to the `Bottom` structure of the ladder queue. Figure 9 provides the pseudo-code for our implementation of the Ladder Queue.

This deviation from the original ladder design calls for handling of the situation where the number of events in `Bottom` exceeds a certain `threshold`. Under this circumstance,

as a design compromise, we allow the algorithm to ignore the specified `threshold` for number of events in `Bottom`.

The results from the tests we performed in threaded WARPED using Ladder Queue, STL Multiset and Splay Tree on a single LTSF queue respectively are presented in Figures 10, 15 and 20. Figures 10 and 15 clearly point towards superior performance of Ladder Queue for most configurations of the simulation. However, Figure 20 does not point in the same direction as the other two figures. Here the Ladder Queue seems to perform on par with Splay Tree and STL multiset until the number of worker threads reaches 32, after which its performance degrades. The degradation in Ladder Queue's performance on transition of the number of worker threads from 32 to 48 is likely due to thread swapping on one or more of the processor cores. The threaded WARPED kernel requires a manager thread in addition to the configured number of worker threads(here 48). The hardware platform in use has 48-cores only. The ISCAS-85 benchmark circuits are a recent addition to our inventory of simulation models. We are still in the process of understanding the nature of event sets that are being generated by these circuits. We suspect that Ladder queue's performance might be getting affected by improper distribution of events in the rung structure. Locking individual rungs, rather than the entire Ladder Queue data structure, is one promising approach for performance enhancement we plan to incorporate in our future work.

5.3 Experiments Combining Ladder and Multi-LTSF Queues

This section combines the work from the previous two sections and explores various configurations using the different data structures underlying the LTSF queue with multiple instances of the LTSF queue. Throughout the remainder of this section, each performance plot shows the simulation time for a fixed number of LTSF queues, and varies the number of threads to show the performance as the number of threads is increased. The number of LTSF queues used is given in the figure caption.

Figures 10, 11 and 15 show that the ladder queue is faster under most circumstances. This can likely be attributed to the proper distribution of events in the `rung` structure and O(1) complexity of the ladder queue.

Figures 12, 16, 17, 20 and 21 show the ladder queue as being only slightly faster or similar in performance. This is likely due to thread-level contention or uneven distribution of events in the `rung` structure during one or more `epochs`. We suspect that migrating the locking mechanism in Ladder Queue to the individual `rungs` would lead to superior performance of Ladder queue for these simulations. This is one avenue we plan to explore further.

Figure 22 shows the Splay tree outperforming Ladder Queue when the number of worker threads is set to 16. This sudden poor performance of the Ladder Queue might well be another example of contention issues.

Figures 13, 14, 18, 19, 23 and 24 display somewhat erratic performance for 8 and 16 LTSF queues. This is likely due to the upper limit for the ideal number of LTSF queues. Dynamic load balancing between the LTSF queues, as explained in sections 5.1 and 6, could partially reduce the effects of this. Further analysis will be necessary to fully understand what is occurring.

Many plots show an interesting simulation time convergence at 32 threads and subsequent divergence for 48 threads. As explained at the end of section 5.2, it is likely due to thread swapping on one or more of the processor cores.

6. DISCUSSION

This study pursues two main issues with the pending event lists in the above described implementation of threaded WARPED. The first part of this study is to address the contention when accessing the LTSF queue. We address this problem by replicating the LTSF queue and partitioning the worker threads and the LPs to one of the LTSF queues (Figure 5). This proposed design solution is setup so that the pending event lists for each LP are maintained as independent lists with individual locks. The lowest timestamped event from each LP event list is placed in one of multiple LTSF pending event queues. The (locked) LTSF queues are sorted and used by the *worker threads* to schedule the next event for execution. The worker threads are assigned to a specific LTSF queue and only retrieve work from that LTSF queue. After dequeuing and processing an event from the LTSF, each worker thread then accesses the pending event set of the LP corresponding to the event just executed and removes the next least-timestamped event for insertion back into the LTSF queue. While this approach helps the problem, the challenge of partitioning the work so that the entire system stays in balance and processes events along the critical path of time becomes more problematic. Fortunately, the software architecture used in this study should setup a streamlined mechanism to implement a lightweight load balancing algorithm as described below.

6.1 Load Balancing between LTSF Queues

Conceptually, the manager thread will monitor the system progress and balance to determine if and when a load balance event occurs. The Time Warp mechanism is unique in that events are aggressively processed which complicates the issue of determining which processes are working effectively and which are not. Fortunately the problem of identifying a "useful work" metric for the LPs of a Time Warp simulation has been previously studied [4] [13] [17]. We anticipate exploring these and other metrics to determine the effective work being done from a load balancing perspective. The worker threads can then be triggered by the manager to move LPs from one LTSF queue to another as shown in Figure 25. This mechanism of load balancing is somewhat similar to the top/bottom-halves based kernel-level synchronization discussed in [24]. In addition to these ideas with LTSF queues, we also believe that the ladder queue data structure presents some key opportunities. This is outlined below.

6.2 Ladder Queue Possibilities

Much like lazy queues [19], ladder queues [22] are related to the calendar queue data structure [3]. While the ladder queue presents some interesting properties that can be readily exploited for further optimizing parallel simulation (described below), all of the calendar queue mechanisms present an opportunity for implementing a finer-grained locking mechanism that is suitable for use on many-core processing systems. In particular, all of these mechanism present a hierarchical structure of buckets that can be independently locked for the insertion and deletion of events. The key advantage of ladder queues are the dynamic sizing of the time windows for the buckets used to hold events.

```
if (new event insertion location is first non-empty bucket of last rung) then

  if (first non-empty bucket of last rung exceeds threshold) then

    /* trigger for new rung creation */
    if (bucket width of last rung equals 1) then
      transfer events from that bucket to bottom
      insert the new event directly into bottom

    else

      /* calculate bucket width of new rung */
      bucket width of new rung =
        bucket width of previous rung / max bucket length
      transfer events from that bucket to the new rung

      /* new rung now becomes the last rung */
      insert the new event in the appropriate bucket of new rung

    end if

  else /* first non-empty bucket of last rung does not exceed threshold */

    insert new event into the bucket

  end if

else /* new event insertion location somewhere else */
  continue with faithful implementation

end if
```

Figure 9: Ladder Queue with the modifications introduced for threaded warped. Many details have been omitted for sake of clarity.

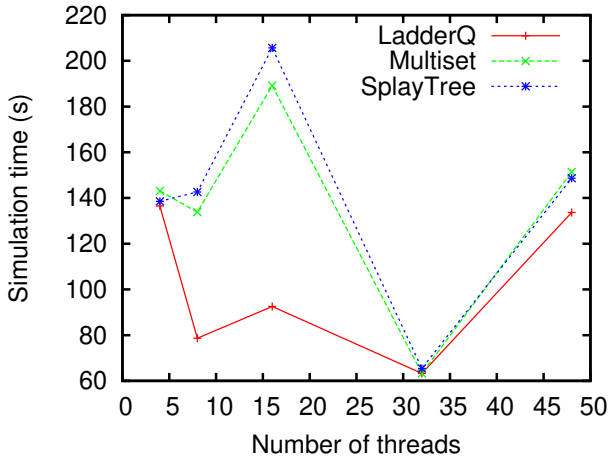

Figure 10: Performance of Different LTSF Data Structures using 1 LTSF Queue with RAID-5

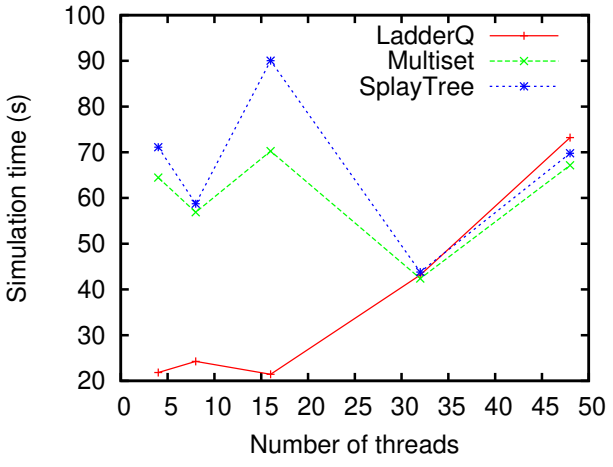

Figure 11: Performance of Different LTSF Data Structures using 2 LTSF Queues with RAID-5

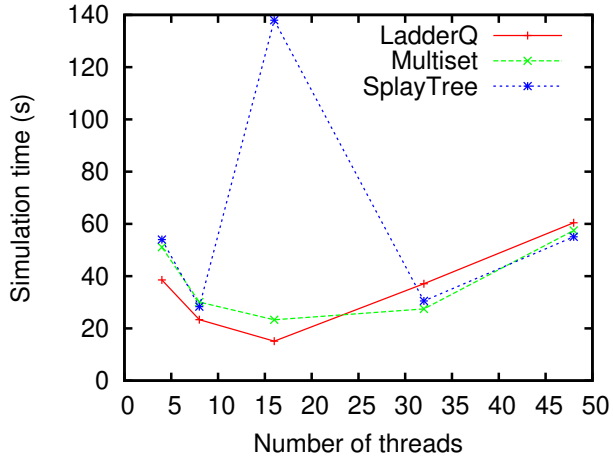

Figure 12: Performance of Different LTSF Data Structures using 4 LTSF Queues with RAID-5

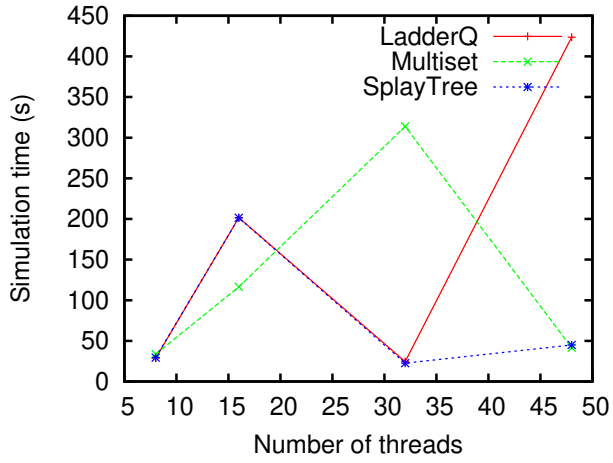

Figure 13: Performance of Different LTSF Data Structures using 8 LTSF Queues with RAID-5

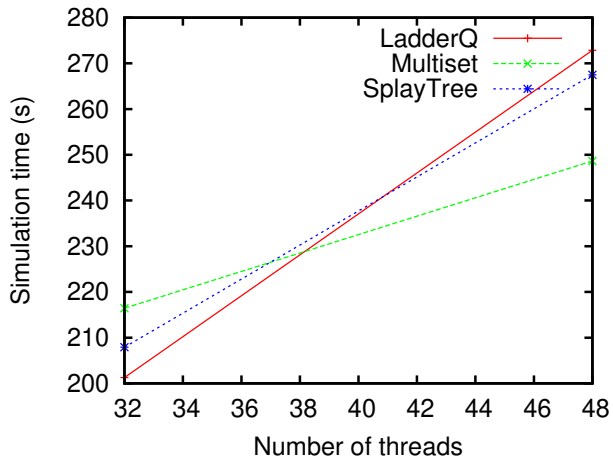

Figure 14: Performance of Different LTSF Data Structures using 16 LTSF Queues with RAID-5

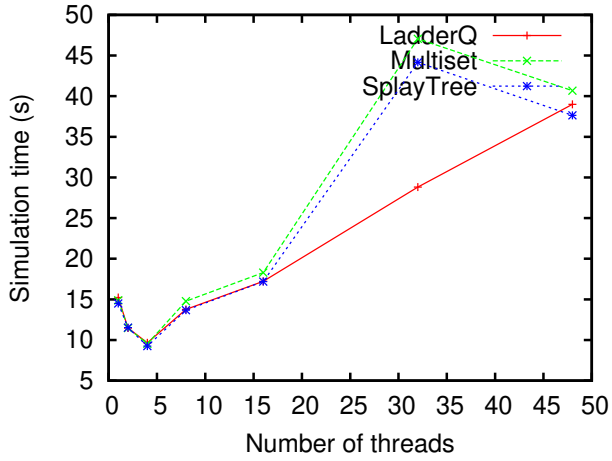

Figure 15: Performance of Different LTSF Data Structures using 1 LTSF Queue with c2670

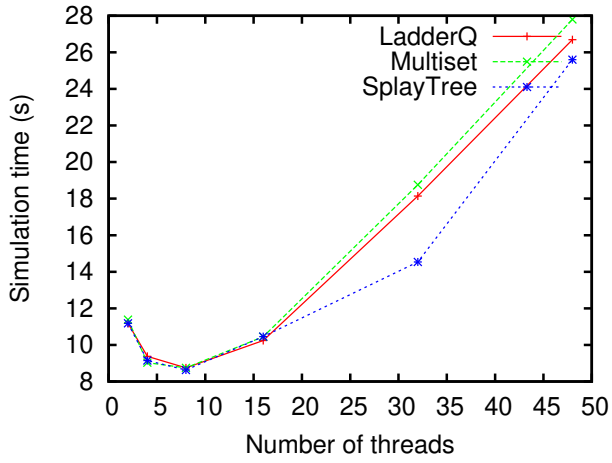

Figure 16: Performance of Different LTSF Data Structures using 2 LTSF Queues with c2670

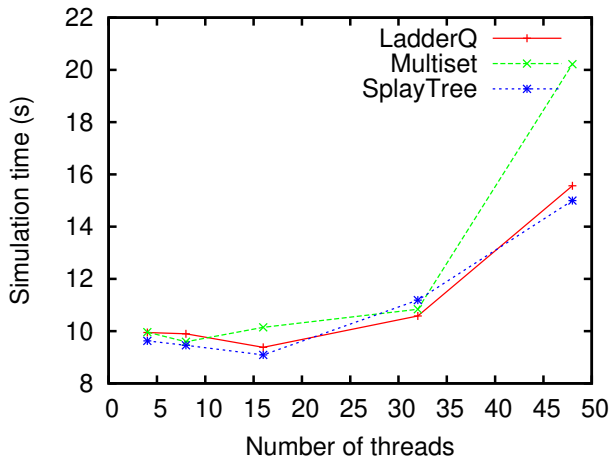

Figure 17: Performance of Different LTSF Data Structures using 4 LTSF Queues with c2670

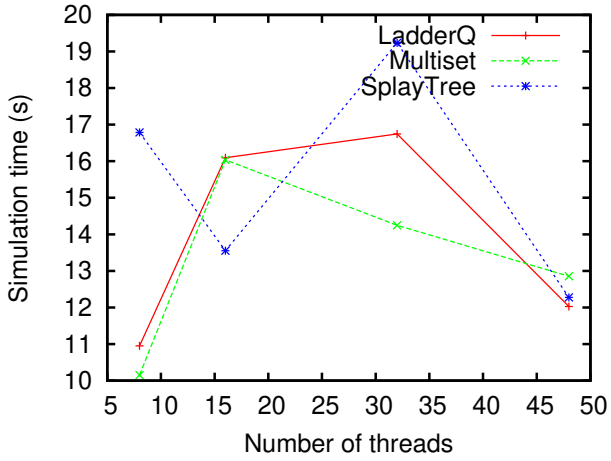

Figure 18: Performance of Different LTSF Data Structures using 8 LTSF Queues with c2670

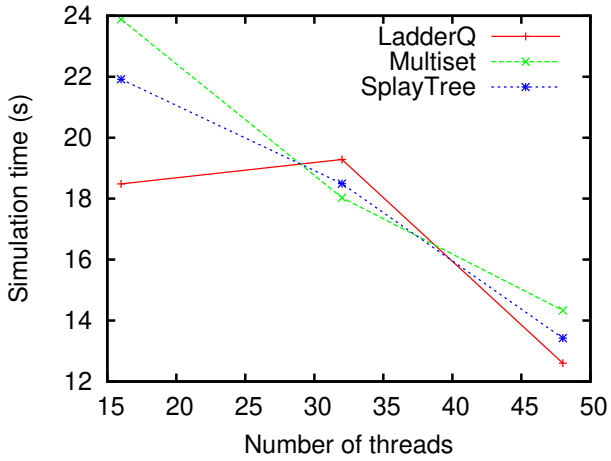

Figure 19: Performance of Different LTSF Data Structures using 16 LTSF Queues with c2670

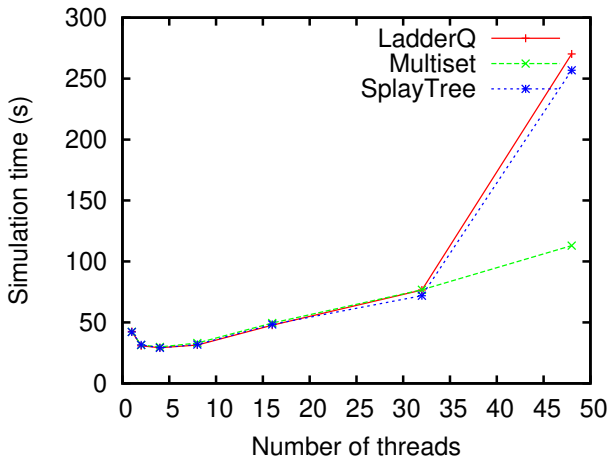

Figure 20: Performance of Different LTSF Data Structures using 1 LTSF Queue with c7552

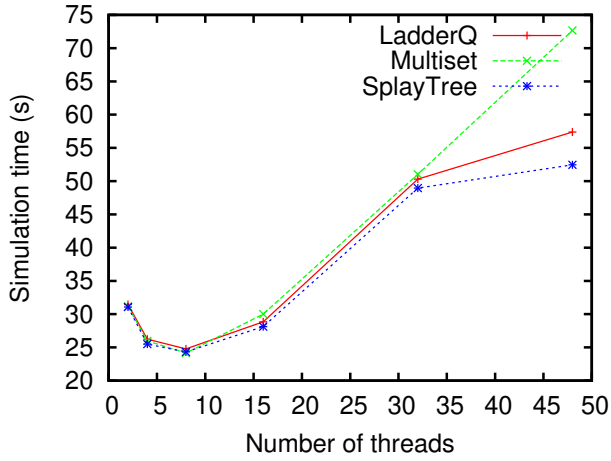

Figure 21: Performance of Different LTSF Data Structures using 2 LTSF Queues with c7552

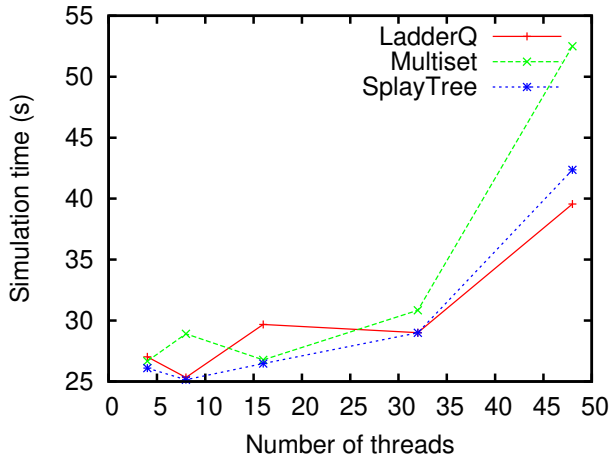

Figure 22: Performance of Different LTSF Data Structures using 4 LTSF Queues with c7552

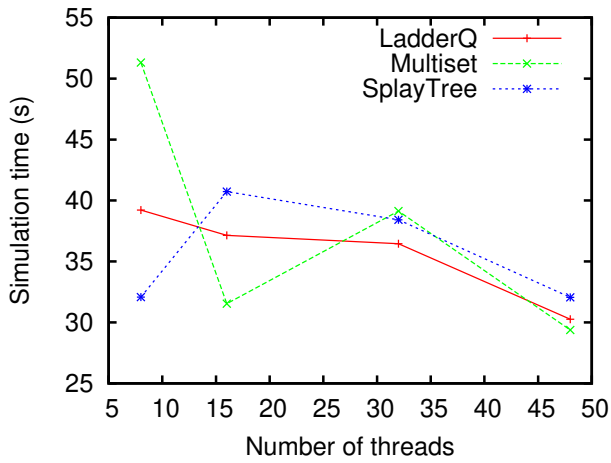

Figure 23: Performance of Different LTSF Data Structures using 8 LTSF Queues with c7552

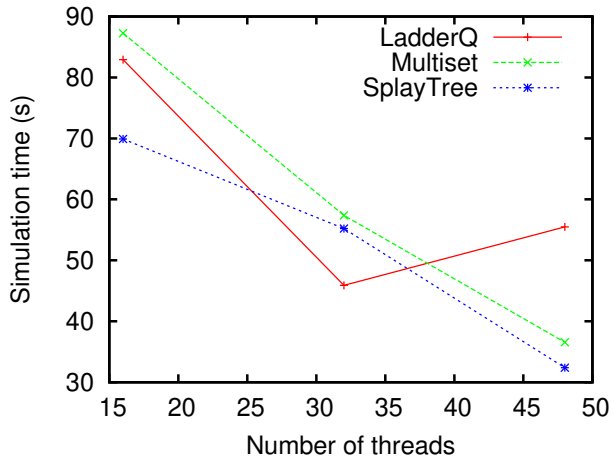

Figure 24: Performance of Different LTSF Data Structures using 16 LTSF Queues with c7552

```
worker_thread_at_LTSFi()
    lock LTSFi queue
    dequeue smallest event from LTSFi
    unlock LTSFi queue
  while !done loop
    process event (assume from LPk)
    lock LPk queue
    dequeue smallest event from LPk
    if loadBalanceRequested then
      lock LTSFj (new target) queue
      insert event from LPk
      unlock LTSFj queue
      unlock LPk queue
      lock LTSFi queue (get next event)
      dequeue smallest event from LTSFi
      unlock LTSFi queue
    else
      lock LTSFi queue
      insert event from LPk
      dequeue smallest event from LTSFi
      unlock LTSFi queue
    end if
  end loop
```

Figure 25: The addition of load balancing/sharing into the worker threads.

While these bucket queue data structures have an O(1) average access time and have been proposed for use in managing the pending event lists in discrete event simulation, the ladder queue has some interesting advantages that we propose are highly desirable for implementing the pending event lists in Time Warp synchronized parallel simulation. In both calendar queues and lazy queues, the bucket size is set to a specific size that may occasional require a resizing operation to manage the time windows sizes. In contrast, the ladder queue has a builtin structure to manage the bucket sizes as per the time density of the incoming events.

We believe that we can further relax the definition of the `Bottom` component of the ladder queue so that it becomes an unsorted list of elements that are accessed using a fully wait-free mechanism achieved through atomic move instructions. Briefly elements are appended and removed in a queue structure without regard to a strict enforcement of time order. The principle idea is that the events in any one bucket are causally independent and therefore a full sorting of these events is unnecessary. Furthermore, because Time Warp maintains the ability to rollback and recover whenever a causal violation occurs, the assumption of causal independence is not catastrophic to the simulation when it fails to hold. The system would simply rollback and reprocess the event after the causally dependent parent event is processed. Of course this solution may not work for simulation models with tight causal relations. That said, this failure will occur only for simulation models where the Time Warp mechanism would not generally succeed anyway and thus, no loss of generality in the use of a Time Warp synchronized parallel simulation solution occurs.

The key advantage of the ladder queue is that the structure is relatively independent on the time granularity of the simulation model operating on top of the simulation kernel. Ideally, the dynamic range of event timestamps for the limited range of local events that define each epoch of the ladder should help ensure that the events within any bucket of the ladder queue are causally independent. This idea is related to the lookahead property of conservatively synchronized parallel simulation [5]. The lookahead concept results from a static analysis of the simulation model that guarantees a time window ahead of any source timestamp in which no additional events will be generated (effectively a minimum time delay guarantee on the event processing latency). While not as strong as lookahead, a key hypothesis for our work is that *the bucket size for the ladder queue will typically encompass a range of event timestamps that are causally independent.* This hypothesis falls from the observation that time ladder queue is reasonably stationary in the time window, t to $t+\Delta t$, for incoming events that arrive in the epochs when events are pulled from `Top` into the ladder (`Rungs` and `Bottom`) of the ladder queue. This causal independence is the principle rationale for the proposed unsorted, wait-free, generalization of the ladder queue proposed herein.

7. CONCLUSIONS

The work described in this paper introduces the ability to use multiple LTSF queues in an attempt to reduce the contention when large numbers of threads are used for the threaded WARPED simulation kernel. Preliminary results show definite improvements for all numbers of threads when moving from one to two LTSF queues, and slight improvements as the number of queues is increased up to some

maximum depending on the number of threads used. The ISCAS-85 simulation results show that the optimal configuration contains between four and eight threads assigned to each LTSF queue, which confirms the value found by Muthalagu [12].

The use of Ladder Queues for the LTSF queues shows an increase in simulation performance for most configurations, performing at least as well, and even significantly better in some circumstances. Splay trees show performance that closely matches that of multiset, and even performs slightly worse in some situations. Based on the results from these simulations, Ladder Queues are an interesting option, especially if we take advantage of the ability to relax the definition of the BOTTOM data structure to make it unsorted.

Once this kernel is further stabilized, we hope to run further experiments using other simulation models to see the effects of these new data structures, and the introduction of multiple LTSF queues. These results show that both Ladder Queues and multiple LTSF queues have performance benefits, but further research needs to be performed into how best to exploit these benefits.

8. ACKNOWLEDGMENTS

Support for this work was provided in part by the National Science Foundation under grant CNS–0915337. We also extend our thanks to Xinyu Guo for implementing the `ISCAS-85` benchmark circuit simulation models.

9. REFERENCES

[1] H. Avril and C. Tropper. Clustered time warp and logic simulation. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 112–119, June 1995.

[2] P. D. R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec. 1972.

[3] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, Oct. 1988.

[4] R. Child and P. A. Wilsey. Using DVFS to optimize time warp simulations. In *Proceedings of the 2012 Winter Simulation Conference*, July 2012.

[5] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[6] A. Ghuloum. Face the inevitable, embrace parallelism. *Communications of the ACM*, 52(9):36–38, Sept. 2009.

[7] M. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test*, 16(3):72–80, July-Sept 1999. (benchmarks available online at: `http://web.eecs.umich.edu/~jhayes/iscas.restore/`).

[8] Intel Press Release, Intel Corporation. Futuristic intel chip could reshape how computers are built, consumers interact with their pcs and personal devices. Technical report, Intel Press Release, Intel Corporation, Dec. 2009.

[9] R. Kalla. Power7: Ibm's next generation power microprocessor. In *Hot Chips 21*, Aug. 2009.

[10] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp simulation kernel for analysis and application development. In H. El-Rewini and B. D. Shriver, editors, *29th Hawaii International Conference on System Sciences (HICSS-29)*, volume Volume I, pages 383–386, Jan. 1996.

[11] R. Miller. Optimistic parallel discrete event simulation on a beowulf cluster of multi-core machines. Master's thesis, University of Cincinnati, 2010.

[12] K. Muthalagu. Threaded warped: An optimistic parallel discrete event simulator for clusters fo multi-core machines. Master's thesis, School of Electronic and Computing Systems, University of Cincinnati, Cincinnati, OH, Nov. 2012.

[13] A. Palaniswamy and P. A. Wilsey. Parameterized Time Warp: An integrated adaptive solution to optimistic pdes. *Journal of Parallel and Distributed Computing*, 37(2):134–145, Sept. 1996.

[14] S. K. Prasad, S. I. Sawant, and B. Naqib. Using parallel data structures in optimistic discrete event simulation of varying granularity on shared-memory computers. In *IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, pages 365–374, Apr. 1995.

[15] R. Radhakrishnan, L. Moore, and P. A. Wilsey. External adjustment of runtime parameters in Time Warp synchronized parallel simulators. In *11th International Parallel Processing Symposium, (IPPS'97)*. IEEE Computer Society Press, Apr. 1997.

[16] R. Rajan and P. A. Wilsey. Dynamically switching between lazy and aggressive cancellation in a Time Warp parallel simulator. In *Proc. of the 28th Annual Simulation Symposium*, pages 22–30. IEEE Computer Society Press, Apr. 1995.

[17] P. L. Reiher and D. Jefferson. Dynamic load management in the time warp operating system. *Transactions of the Society for Computer Simulation*, 7(2):91–120, 1990.

[18] R. Rönngren, R. Ayani, R. M. Fujimoto, and S. R. Das. Efficient implementation of event sets in time warp. In *Proceedings of the 1993 workshop on Parallel and distributed simulation*, pages 101–108, May 1993.

[19] R. Rönngren, J. Riboe, and R. Ayani. Lazy queue: An efficient implementation of the pending-event set. In *Proc. of the 24th Annual Simulation Symposium*, pages 194–204, Apr. 1991.

[20] T. Santoro and F. Quaglia. A low-overhead constant-time ltf scheduler for optimistic simulation systems. In *Proceedings of the The IEEE symposium on Computers and Communications*, pages 948–953, June 2010.

[21] D. Sleator and R. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[22] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng. Ladder queue: An o(1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation*, 15(3):175–204, July 2005.

[23] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, Apr. 1985.

[24] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 2012 ACM/IEEE/SCS*

26th Workshop on Principles of Advanced and Distributed Simulation, pages 211–220, July 2012.