Lock-Free Pending Event Set Management in Time Warp

Sounak Gupta
Dept of of Electrical Engineering and Computing Systems
Cincinnati, OH 45221-0030
sounak.besu@gmail.com

Philip A. Wilsey
Dept of of Electrical Engineering and Computing Systems
Cincinnati, OH 45221-0030
wilseypa@gmail.com

ABSTRACT

The rapid growth in the parallelism of multi-core processors has opened up new opportunities and challenges for parallel simulation discrete event simulation (PDES). PDES simulators attempt to find parallelism within the pending event set to achieve speedup. Typically the pending event set is sorted to preserve the causal orders of the contained events. Sorting is a key aspect that amplifies contention for exclusive access to the shared event scheduler and events are generally scheduled to follow the time-based order of the pending events. In this work we leverage a Ladder Queue data structure to partition the pending events into groups (called buckets) arranged by adjacent and short regions of time. We assume that the pending events within any one bucket are causally independent and schedule them for execution without sorting and without consideration of their total time-based order. We use the Time Warp mechanism to recover whenever actual dependencies arise. Due to the lack of need for sorting, we further extend our pending event data structure so that it can be organized for lock-free access. Experimental results show consistent speedup for all studied configurations and simulation models. The speedups range from 1.1 to 1.49 with higher speedups occurring with higher thread counts where contention for the shared event set becomes more problematic with a conventional mutex locking mechanism.

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

Keywords

Time Warp, pending event lists, multi-core, threads, lock-free

1. INTRODUCTION

The adoption of parallelism through core replication to produce multi-core and many-core processors is widespread and growing. Inexpensive processors with core counts of 4–8 are common. At a slightly higher (but still affordable) price point, core counts of 12–16 are readily available. Furthermore, there is every expectation that these numbers will continue to increase. The introduction of these multi-core and many-core processors into mainstream Beowulf Clusters means that distributed algorithms must be developed that support parallelism both within and between the nodes of the cluster [6]. Shared data must be represented and organized to support high speed, access with a minimum of contended accesses by the parallel threads that need to read and update the shared data. This is especially true for fine-grained applications such as parallel discrete event simulation.

The central data structure on a discrete event simulator is the pending event set. This set records and organizes the events yet to be processed by the simulation kernel. For a parallel simulation kernel with multiple threads accessing a shared pending event set, the organization and management of the pending event set must be carefully planned. The pending event set is frequently accessed as the event processing threads dequeue events for processing and enqueue any generated events (for purposes of this manuscript, we will ignore the details of remote event transmission to other nodes of a Beowulf Cluster). Furthermore, since the processing of discrete events is often a fine-grained computation, significant contention for the protected pending event set can rapidly grow to negatively impact performance. As few as 5-6 threads can easily result in performance loss triggered by contention [18].

This manuscript examines the management of a shared pending-event set for a Time Warp [9, 5] synchronized parallel simulation kernel executing on a single node many-core processing platform. The work builds on the two-level approach for managing the pending event set that was originally developed by Karthik [18] and extended by Dickman *et al* [4]. The extension by Dickman *et al* used the ladder queue data structure [29] to manage the scheduling event queue. The ladder queue is a variant of the calendar queue [3] that arranges events into *months* (or *buckets*) so that their timestamp falls within a bounded time window assigned to that bucket. Thus, the events in each bucket are guaranteed to fall within a small window of time. In the ladder queue, these buckets are not sorted (to save time) until the actual dequeue operations occur on a particular bucket.

At the end of their experimental analysis section, Dickman *et al* suggest that the small time window for the pending events in a particular bucket may be such that the contained events are mostly causally independent. Should this be true, in a Time Warp synchronized parallel simulation, one could manage all of the buckets in the ladder queue as unsorted queues and potentially use atomic operations to build a lock-free data structure for managing the pending event set. Given the relatively high costs of locks or other mechanisms to provide a critical region of access, a lock-free pending event set could substantially reduce access time and contention to this critical shared resource. Time Warp is a critical contributor to this possibility since it can recover, by rollback, should a causal dependency actually be experienced when processing the unsorted events from the ladder queue bucket. In this manuscript, we examine this idea more fully and show experimental results for a parallel Time Warp simulator with a lock-free unsorted ladder queue data structure for managing the pending event set.

The remainder of this manuscript is organized as follows. Section 2 provides a brief review of studies related to pending event set and non-blocking, lock-free lists/queues. Section 3 provides details about the software architecture of WARPED, the Time Warp simulation kernel which has been used in this work. Section 4 describes our proposed non-blocking ladder queue design in details. Section 5 presents the results of our experimental analysis. Section 6 is a discussion about the possibilities of further change in the pending event set design. Finally, Section 7 presents the conclusions we can draw from this experiment.

2. RELATED WORK

2.1 Pending Event Sets

WARPED is a parallel simulation kernel that implements the Time Warp synchronization protocol [12, 23]. Management of the pending event set in WARPED follows the model outlined in [26] and decomposes it into `Unprocessed` and `Processed` event pools. The `Unprocessed` pool stores the events that are yet to be executed. The `Processed` pool refers to those events that have been processed but not yet committed (they must be preserved in case of rollback). As per the Time Warp protocol, WARPED has been designed to greedily process events without strict adherence to causal order. Whenever causal violations are detected, WARPED rolls back to a consistent state (usually the last commit point) and re-executes the events in their proper order. To support this rollback, the `Processed` queue serves as the holding list for events that might need re-execution in case of a rollback [4, 18]. Similar models have been adopted for other time warp synchronized simulators. For example, a simple implementation of a doubly-linked list to store all processed and unprocessed events along with their execution status was proposed by Ronngren *et al* [26]. Although this design allows quick and efficient rollbacks and `fossil` collection, it cannot effectively insert and delete events when the `Unprocessed` event pool is large. An improved skew heap was suggested by Ronngren as a possible data structure that can be effective for large `Unprocessed` event pools.

A medium to coarse-grained simulator was built by Prasad *et al* [22] using a parallelized Calender Queue. They provided each processor with a different calender queue. Their study showed that load balancing of a local queue-based simulator was comparable to that of a global queue-based simulator. However, the global queue-based simulator was faster with fewer rollbacks. Using an array and hierarchical bitmap, Santoro *et al* [27] created a Least-Timestamp-First (LTSF) scheduler. Although similar to a Calendar Queue, this low-overhead scheduler provides a constant access-time.

Dickman *et al* [4] proposed the use of multiple LTSF queues in scheduler to reduce contention on a many-core processing platform. A comparative analysis also showed that `Ladder Queue` [29] performed better as a LTSF queue in WARPED scheduler compared to `STL multiset` (a sorted doubly-linked list) and `Splay tree` [28].

2.2 Lock Free Lists

Valois [31] proposed the first lock-free list that required only atomic compare-and-swap (CAS) operations. He used the technique of encoding in-progress operations in auxiliary nodes. This technique was improved by Michael *et al* [15]. A lock-free ordered list was implemented by Harris [7] using pointer marking technique. The least significant bit of the next pointer of a deleted node is marked to denote logical deletion. Physical removal of that node takes place in a separate phase. Memory reclamation in Harris' algorithm was improved by Michael [13] using hazard pointers [14]. A lock-based linked list was designed by Heller *et al* [8] that used wait-free look up operations. Harris and Michael used this wait-free approach to improve the performance of their algorithm. A wait-free queue was proposed by Kogan and Petrank [10]. Their fast-path-slow-path method is composed of a slower wait-free algorithm coupled with a comparatively faster lock-free algorithm. This approach shows better performance than Harris-Michael algorithm. Timnat *et al* used ideas from [30] to design a wait-free ordered linked list. They used Harris-Michael algorithm as the fast path.

Liu *et al* [11] proposed a lock-free scheduler for conservative parallel simulation. Their implementation uses `fetch-and-add` atomic operations to improve performance on a shared-memory multiprocessor platform. They observed that with increases in the number of logical processes, the performance improvement was marginal.

3. BACKGROUND: EVENT MANAGEMENT AND PROCESSING IN WARPED

WARPED is a parallel discrete event simulation kernel that implements the Time Warp synchronization protocol [12, 23]. Initially designed and optimized for parallel simulations on single core processor-based Beowulf Clusters, it incorporates extensive configurable features and sub-algorithms of the Time Warp Mechanism (*e.g.,* adaptive periodic checkpointing [19] and lazy, aggressive, and dynamic cancellation [25]). On each processing node, the Logical Processes (LPs) of a simulation are grouped together and scheduled according to a Least-Timestamp-First (LTSF) event scheduling policy. The Time Warp housekeeping functions such as GVT estimation, termination detection, and fossil collection are organized into a set of common services for the entire LP population on that node. This node-based architecture is similar to that reported in [1] and [24].

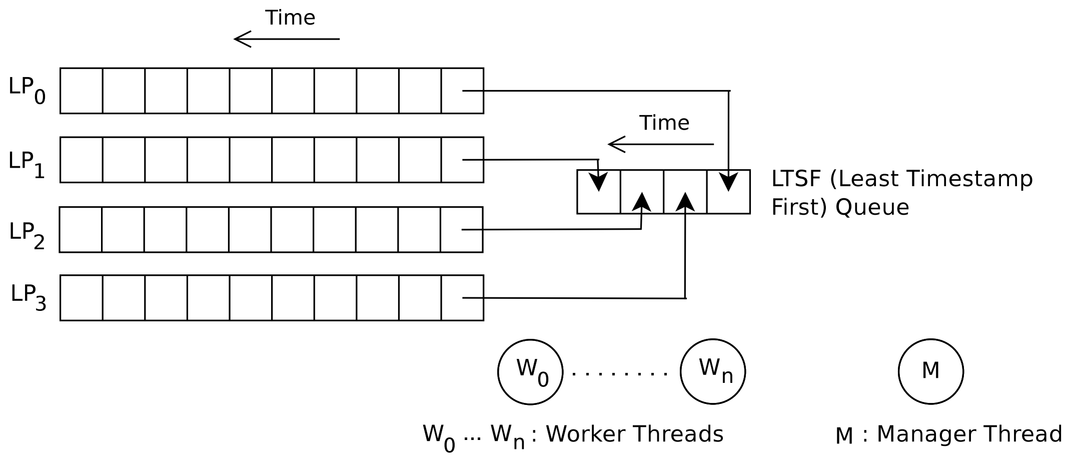

Time
LP0
LP1
LP2
LP3
Time
LTSF (Least Timestamp First) Queue
W0
Wn
M
W0 ... Wn : Worker Threads
M : Manager Thread

Figure 1: Threaded structure of warped.

Recently, WARPED was extended to incorporate threaded execution for multi-core and many-core processors and Beowulf Clusters composed of such [17, 18]. Figure 1 shows the overall design structure consisting of the main pending event pool and the executing threads. The initial design worked reasonably well for smaller multi-core processor systems. A high level overview of the thread operation and pending event set organization is provided below.

A manager thread and one or more worker threads constitute a threaded instance of WARPED. The Time Warp housekeeping functions are processed by the *manager thread* (labeled `M` in Figure 1). The receipt and transmission of event messages exchanged with remote nodes in the cluster is also handled by the *manager thread*. However, the local event insertion is performed by the worker threads. Additional details on the operation of the manager thread design are provided in [18]. The dequeueing and execution of pending events and the subsequent generation of new events is handled by the *worker threads* (depicted as $W_0 \cdots W_n$ in Figure 1).

The pending event sets are organized into a two level structure. At the first level, each LP managed a separate sorted linked list of its pending events. These lists are locked and accessed by the manager and worker threads. At the second level is (one or more) common LTSF pending event queue where the lowest timestamped event from each LP event list is enqueued. The *worker threads* use the (locked) LTSF queue to schedule the next event for execution. After dequeueing and processing an event from the LTSF queue, a worker thread will replenish the LTSF queue by removing the new lowest timestamped event from the pending event list of the LP corresponding to the event just processed and insert it into the LTSF queue. The algorithm shown in Figure 2 provides a pseudo code representation of the general event processing performed by the worker threads.

When configured with only a few worker threads, this system works well. However, once the number of worker threads exceeds 5–6, there is a negative impact on the performance due to contention for the LTSF queue. Contention is not a problem for the LP event pools as these structures are independently locked and only one worker thread and the manager thread can simultaneously access the same LP event pool. The LTSF queue is the focal point of contention for pending events in this architecture. The organization of the pending event list needs modification, especially the LTSF queue.

```
worker_thread()
    lock the LTSF queue
    dequeue the smallest event from the LTSF
    unlock the LTSF queue

    while !done loop
        process event
            (assume this event belongs to LP[i])
        lock LP[i] queue
        dequeue smallest event from LP[i]
            (assume this event to be k)
        lock the LTSF queue
        insert event k into the LTSF
        dequeue the smallest event from the LTSF
        unlock the LTSF queue
        unlock LP[i] queue
    end loop
```

Figure 2: Generalized event execution loop for any worker thread.

The principle solution to the contention issue in threaded WARPED kernel is to support multiple LTSF queues (Figure 3). The worker threads are uniformly (or near uniformly) divided into a number of independent groups. The number of such groups equals the number of LTSF queues desired. Each worker thread group is statically assigned to a LTSF queue. Figure 3 illustrates the binding of worker thread groups to LTSF queues (the worker threads are denoted by the bubbles labeled $W_0 \cdots W_n$). Correspondingly, the LPs are divided into a number of independent groups bound to specific LTSF queues. The LTSF queues are then populated with events from their assigned LPs in a manner similar to the single LTSF implementation. The assignment of worker thread groups is designed to be fixed throughout the simulation. LP groups, however, can be reassigned among the LTSF queues dynamically during simulation to facilitate

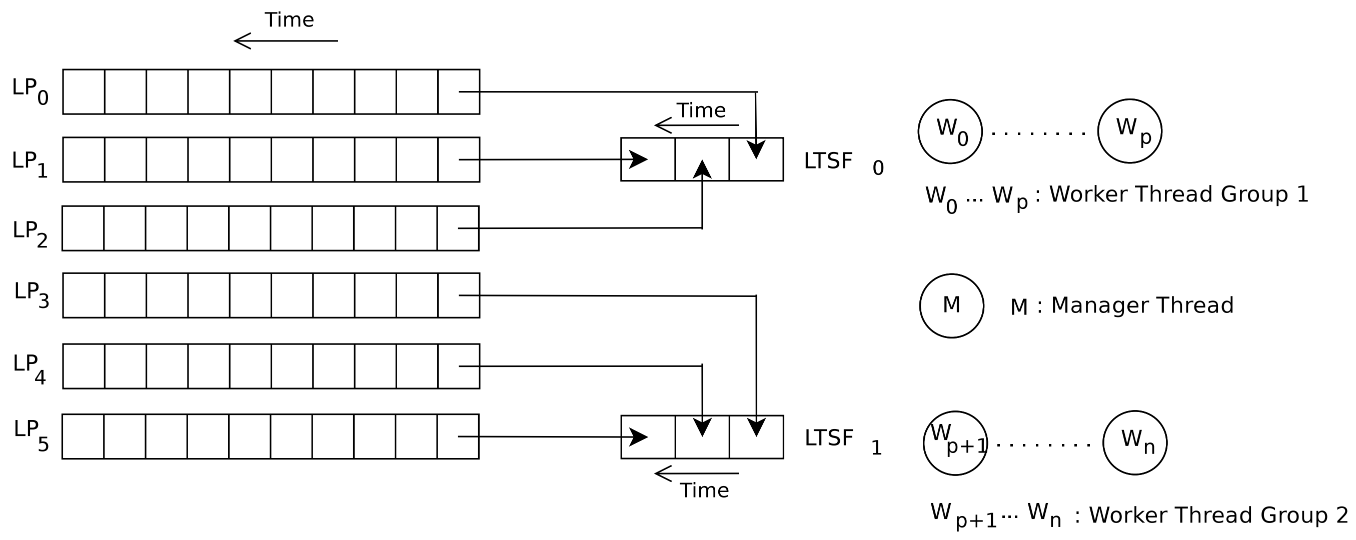


Figure 3: Threaded structure of warped with multiple LTSF queues.

load balancing (or more precisely to distribute the critical path of events for processing) [4].

Partitioning the LPs into groups, each serving as event pool to a specific LTSF queue, does improve the performance due to reduction in contention for individual LTSF queues. However, this re-organization can prove to be a "double-edged sword". The basic problem with this organization is the challenge of statically partitioning the LP groups. The unique aspect of the Time Warp scheduling is that events are aggressively processed and the system is rolled back to a consistent state when any inconsistency is detected. This makes it difficult to determine which processes are working effectively and which are not. One approach is to "kick-start" the simulation with arbitrary initial partitions of the LPs followed by intermittent monitoring and rectification of the imbalance through load balancing [4].

The underlying data structures for Implementing the LTSF queue were also explored [4]. In these studies, the `Ladder Queue` [29] delivered superior performance over both the `STL multiset` and `Splay tree` [28] implementations. The significant characteristic of the `Ladder Queue` (also its principal difference with `Calendar Queue` [3]) is the dynamic splitting of buckets (months) that store events. When the number of stored events exceeds some threshold, `Calendar queue` requires a dynamic resizing of the entire data structure. In contrast, the `Ladder queue` dynamically splits a bucket (when it exceeds some specified threshold) into a collection of buckets and therefore requires no resizing operations. The principal components of a `Ladder Queue` are shown in Figure 4. An overview of the operation of a `Ladder Queue` is outlined below.

Initially the ladder queue is empty. Incoming events are inserted into the `Top` component in order of their arrival (not sorted based on event timestamp). While receiving events into `Top`, the minimum and maximum timestamps of the events placed therein are recorded. On receiving the first dequeue request, the events in `Top` are transferred to a collection of buckets in the first rung of the ladder (`Rung[1]`). Each bucket in `Rung[1]` holds events for the timestamp range that equally divides the time range between the minimum and maximum timestamp of events that were originally stored in `Top`. Each event is placed, without sorting, into the bucket encompassing its timestamp value. There is an upper threshold on the number of events in each bucket. If the first non-empty bucket of the rung exceeds this threshold, a new lower `Rung` is defined and events from the overflowing bucket are transferred to the collection of buckets in the next lower rung (`Rung[2]`). This redistribution is illustrated in Figure 4.

To complete the dequeue operation, the events from the first non-empty bucket (containing the elements with timestamp ranges smaller than all of the remaining buckets) of the lowest rung are *sorted* and placed in `Bottom`. The first event (having the lowest timestamp) is then dequeued from `Bottom`. The events are pulled from `Bottom` for successive dequeue operations until it becomes empty. This condition triggers another pull of events from the first non-empty bucket of the lowest rung in the ladder. The dequeue activity continues in this way until there are no more events left in the `Rungs` and `Bottom`. New events from `Top` are then allowed to re-populate the `Rungs` and `Bottom` in the manner discussed above.

In the ladder queue, timestamp values govern the incoming event distribution once the initial ladder structure is populated. The ladder queue then partitions the event timeline into epochs. Events with timestamp value between t and $t + \Delta t$ are held by the `Bottom` and `Rung` structures while the `Top` acts as temporary storage for events with timestamp above $t + \Delta t$. When the dequeue operation empties the `Bottom` and `Rung` contents, another ladder queue epoch occurs. Events in `Top` are transferred to the `Rungs` and `Bottom`. The additional special cases are discussed in [29]. For use in threaded WARPED, the `Ladder Queue` algorithms were modified slightly (see to Section 5.2 of [4]).

4. A LOCK-FREE AND UNSORTED LTSF QUEUE

In our past work [4], a comparative analysis of the original `Ladder Queue`'s performance was presented. In this paper, we discuss two significant modifications to the Ladder Queue

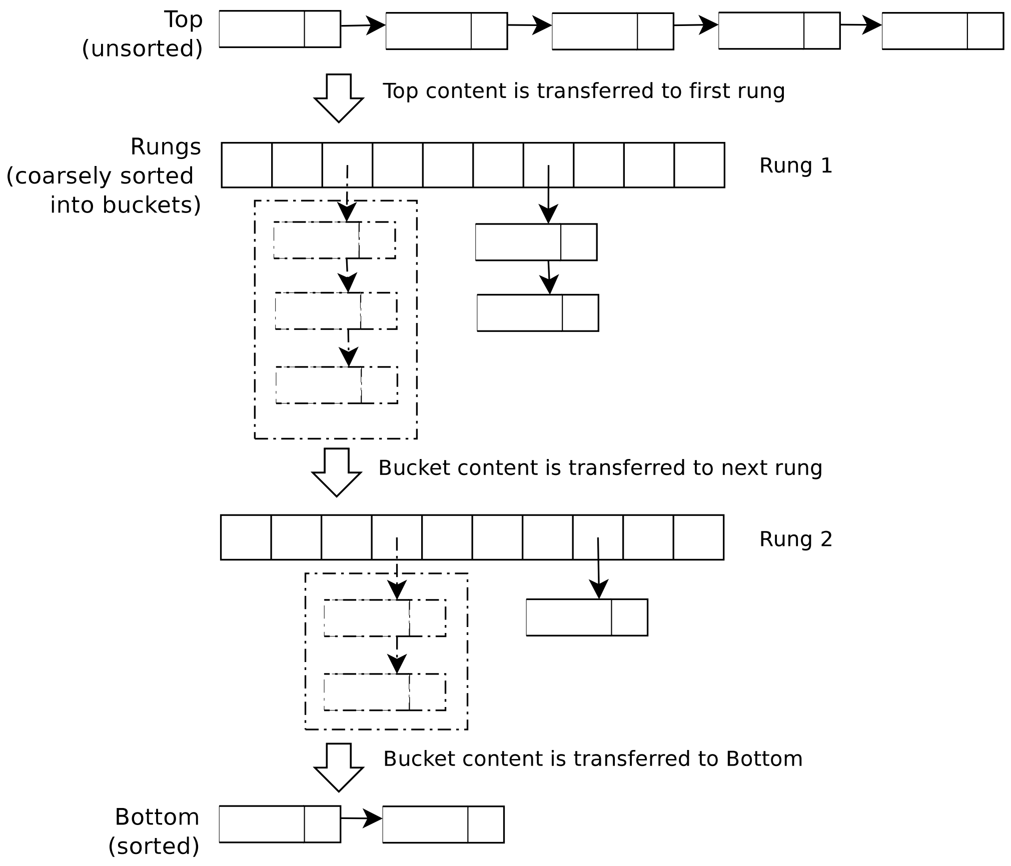

Top
(unsorted)
Top content is transferred to first rung
Rungs
(coarsely sorted
into buckets)
Rung 1
Bucket content is transferred to next rung
Rung 2
Bucket content is transferred to Bottom
Bottom
(sorted)

Figure 4: Illustration of the Ladder Queue Structure

structure to further optimize its use in the WARPED simulation kernel. In particular, we consider:

1. replacing the sorted list in the `Bottom` ladder queue structure with an unsorted list, and
2. replacing the mutex locks used in the management of the structures of the ladder queues with lock-free accesses.

The first modification is motivated by an observation that the limited size of the time window in `Bottom` is such that most events contained there are likely to be causally independent. Since the Time Warp mechanism is non-strict in its adherence to causal order and since it can also recover from causal violations, an accidental out-of-order processing of events is not a catastrophic happening. The simulator can merely rollback and reprocess events in their proper order. Thus, the partitioning of events into buckets (as normally required by the ladder queue) also partitions the events into coarse time windows that can be greedily scheduled by the event scheduler. Ideally, the generation of new events will fall outside the time window of the current bucket and the system will operate without significant rollback and without a full sort of input events. While this style of processing may not work efficiently for all simulation models, those models are also not likely to be suitable for Time Warp in general. Thus, this approach should not be a significant drawback for a Time Warp synchronized parallel simulator. The concept of event partitioning (into ladder queue buckets) and the unsorted processing of events in a bucket, will be referred to as "relaxed order of event causality" (or relaxed causality) in the remainder of this paper. Events scheduled using the original `Ladder Queue` are said to have a "strict order of event causality" (or strict causality).

The opportunity to use a lock-free modification arises because once the need to sort the queues are removed, it becomes possible to use a simpler data structure that can be implemented with high performance lock-free access. The details of the implementation and use of a suitable lock-free data structure in WARPED is described in the next section. There is, however one additional small change that we made to our Ladder Queue implementation. Namely we removed the check to split the `Bottom` queue when additional insertions would normally trigger such a split. This is done to further optimize performance. The revised algorithm that illustrates this change to the implementation is shown in Figure 5.

4.1 The Lock-Free Ladder Queue

The WARPED LTSF queue supports the following operations for an event `e`:

`void enqueue(e)`: insert the event `e` into the LTSF queue,

`event* dequeue()`: return the event from the LTSF queue that has the smallest timestamp, and

`void remove(e)`: remove the event `e` from the LTSF queue.

```
void enqueue() {
    /* Try inserting into Top */
    if( timestamp of new event >= minimum timestamp of event in Top ) {
        insert into Top
        return;
    }

    /* Try to locate a suitable rung */
    while( timestamp of new event < min. timestamp of event in rung[i] &&
                                       i <= available number of rungs     ) {
        i++;
    }

    /* Check if rung found */
    if( i <= available number of rungs ) {
        determine the bucket number (j) suitable for the new event
        insert the event in the bucket j of the rung[i].
    } else {

#ifdef SORTED_BOTTOM
        /* Check if number of events in bottom exceeds threshold */
        if( number of events in bottom > threshold ) {
            create a lower rung (if possible)
            transfer Bottom to this lowest available rung
            insert the new event in an appropriate bucket of this rung
        } else {
            insert into Bottom
        }
#endif

#ifndef SORTED_BOTTOM
        insert into Bottom
#endif

    }
}
```

Figure 5: Ladder Queue enqueue() for unsorted Bottom written in C style. Some details have been omitted.

The `remove(e)` operation is needed so that when an LP receives an event `e` with a timestamp that is smaller than the timestamp for the event at the head of its unprocessed events, the system can replace the entry in the LTSF queue. Migrating to the Ladder queue with an unsorted bottom, these operations are maintained, however, the `dequeue()` event is redefined to simply return the event at the head of `Bottom`; the sorting of events in `Bottom` is no longer maintained.

Since the implementation of a functionally correct lock-free algorithms can be quite difficult, using a existing algorithm is most desirable. Examining the literature, we found two promising candidate solutions, namely: the queue algorithm developed by Michael and Scott [16], and the LFList algorithm developed by Zhang *et al* [33]. To support our needs, the queue algorithm would have to be extended to include the `remove(e)` operation and the LFList algorithm would have to be extended to include an operation to remove the head element. After studying the problem and developing several strategies to extend each algorithm, it became apparent that the simpler (yet still highly efficient) solution would be to adapt and extend the LFList algorithm for our needs. This adaption and extension is described below.

4.2 LFList

The LFList algorithm supports an unsorted doublely-linked list of elements using lock-free compare-and-swap (CAS) instructions. For uncontended cases, the insert operation requires 2 CAS instructions and a remove operation requires 1 CAS instruction. To facilitate lock-free access, the LFList algorithm defines four states for elements in the list, namely: INS (insert), REM (remove), DAT (valid data), and INV (invalid). INS and REM are intermediate states that correspond to nodes that are, respectively being "inserted" and "removed". The DAT state is assigned to a node that has been successfully linked into the list and the INV state is assigned to a node that has been marked for removal. The actual removal of INV states occurs during some later insert or remove operation.

Pseudo code representations of the `INSERT` (enqueue) and `REMOVE` (dequeue) operations are shown in Figures 6 and 8. Graphical representations of a representative list being operated on by the `INSERT` and `REMOVE` function are shown in Figures 7 and 9. In Figure 7 we can see how the inserted node is initially placed in the list in state INS. Upon completion of the insert operation, the state is changed to

```
Node:
  key   : stores the Event pointer
  next  : points to successor
  state: current status of any node
         (insert, remove, data, invalidate)
List:
  head : starting node (initially NULL)

function INSERT( Event *k in, bool out ) :

  1. create a new node with ‘k’ as key and
     ‘insert’ as status

  2. add the new node at the front of list
     and mark it as head using CAS.

  3. check whether key ‘k’ is already
     present in the list; this determines
     the return value. Remove existing
     ‘invalid’ nodes during this search.

  4. using CAS try to change the state of
     the node from ‘insert’ to ‘data’ if
     key is unique in the list;
     else, try to ‘invalidate’ it.

  5. if CAS not successful, clean up the
     invalidated nodes in the list.

end INSERT
```

Figure 6: Non-blocking unsorted list details and insert function. Some details have been omitted.

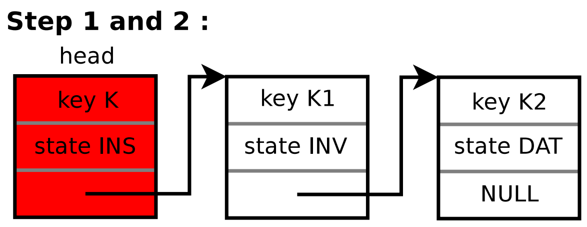
DAT. Note also that the list has a (previously deleted) node marked as state INV. Due to the nature of this lock-free algorithm, the remove operation actually marks nodes with state INV (Figure 9) which is then removed on a later operation (by either INSERT or REMOVE) on the list. More complete details on the operations of LFList are available in [33].

In our implementation of LFList, we removed the backward link (`prev`) and the thread id (`tid`) that was part of the algorithm presented by Zhang *et al* [33]. We did not require a doubly linked list and the thread id was needed for a wait-free derivation of their list that we also did not require. Finally, we also had to define a `dequeue()` operation for the LFList. This is easily achieved by finding an event `e` from the LTSF queue and simply using the `remove(e)` operation of LFList. Pseudo code for the `dequeue()` operation is shown in Figure 10.

5. EXPERIMENTAL ANALYSIS

The purpose of this study is to analyze the effectiveness of causality relaxation in threaded WARPED. The use of an unsorted list as `Bottom` in Ladder Queue [29] allows us to study the overall effect of out-of-order event processing on the simulation. The effect of lock-free operations in the above mentioned Ladder structure is another aspect that has been studied in this paper.

All simulations were run multiple times on the same machine and the results averaged. The machine used for our



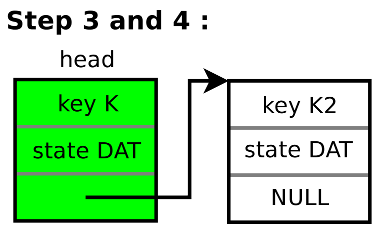
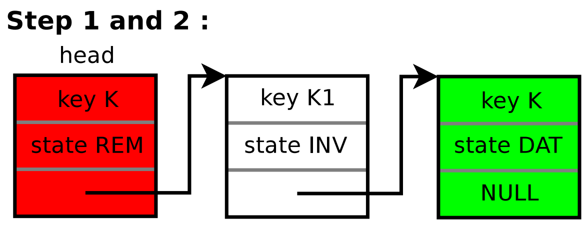


Figure 7: Illustrative example of LFList Insert

```
function REMOVE( Event *k in, bool out ) :

  1. create a new node with ‘k’ as key and
     ‘remove’ as status

  2. add the new node at the front of list
     and mark it as head using CAS.

  3. search for the node with key ‘k’ in the
     rest of the list and ‘invalidate’ it if
     it is in ‘data’ mode; this determines
     the return value. Remove existing
     ‘invalid’ nodes during this search.

  4. move the state of the created node to
     ‘invalidate’.

end REMOVE
```

Figure 8: Non-blocking unsorted list remove function. Some details have been omitted.



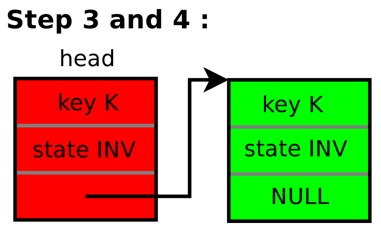


Figure 9: Illustrative example of LFList Remove

```
function DEQUEUE() :

  while the list has nodes with state DAT {
    find the first available node k with state DAT
    if (REMOVE(k) == true) {return k;}
  }
  return NULL;

end DEQUEUE
```

Figure 10: A Dequeue operation for LFList.

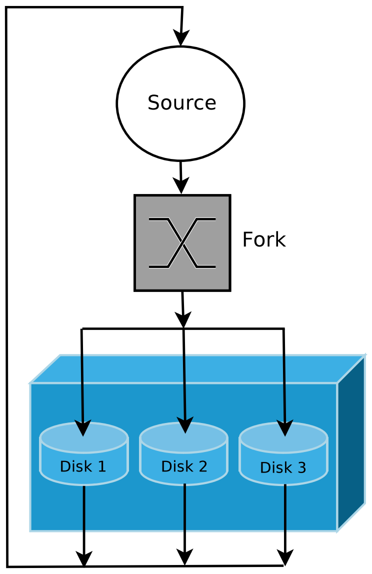


Figure 11: RAID-5 Simulation Model

simulations has 48 cores — four 12-core AMD Opteron 6168 processors, each core running at a clock rate of 1.9GHz. It is configured with 64 GB of RAM.

The following two simulation models were used for the experimental analysis:

RAID-5: This model represents a Level 5 RAID (Redundant Array of Inexpensive Disks) setup. Figure 11 shows the schematics of this model. It consists of 136 logical processes (LPs) that simulate 32 disks, 8 forks and 96 sources. Requests for data from the array are generated by the sources. These requests pass through their respective forks. The forks forward each request to the necessary disks. After a predetermined amount of time, each disk responds to any data request received from the fork. This model simulates the access time to any sector of the disk. The simulation is allowed to continue till the global execution time reaches one hundred thousand seconds.

Epidemic Disease Propagation: This model simulates the spread of disease during an epidemic outbreak. It consists of 56 logical processes each of which simulates a different location. Each location has a group of people, each with different stages of the disease (infected, latent, incubating, infectious, asymptotic or recovered). The progress of disease in a person is controlled by a finite state machine proposed

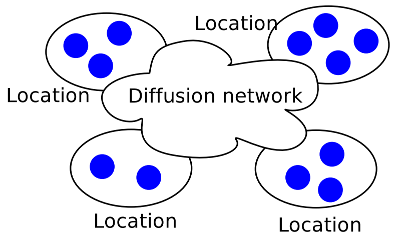


Figure 12: Epidemic Simulation Model

by Barrett *et al* [2]. People can travel between locations at any point during the simulation. There is a pre-defined travel time between any two locations. A reaction-diffusion model based on [20] has been employed to simulate the intra-location and inter-location spread of the disease. The reaction function determines the intra-location spread of disease. The diffusion network models the inter-location movement pattern of persons. Two types of diffusion network have been used, namely:

1. **Fully Connected** : A person can travel to any location in a single hop.
2. **Watts-Strogatz** [32]: This models the small-world behavior of human networks where a person has the option of traveling to any location in a short number of hops.

Similar to RAID-5, the simulation is allowed to continue until the global execution time reaches one hundred thousand seconds. Figure 12 shows the schematics of this model.

Different configurations with varying numbers of worker threads were used for the simulations. The number of worker threads used were 4, 8, 16 and 32. These worker thread configurations were also permuted with varying number of LTSF queues (1, 2 and 4). The number of LTSF queues is always kept less than the number of worker threads and increased by a power of two. This allows even distribution of worker threads for each LTSF queue and helps to keep the simulations balanced. All results were obtained by taking the mean of ten simulation runs. The results from these LTSF queue experiments are described below.

Figures 15, 18 and 21 compare the performance of WARPED scheduler by replacing its sorted `Ladder Queue`-based LTSF with a lock-free `Ladder Queue` having an unsorted `Bottom`. When using a single LTSF queue, we observe significant speedup as the number of threads increase. This is most likely due to role thread contention plays in slowing access to the shared pending event set.

A sorted `Ladder Queue` uses traditional locks to make its operations thread-safe. Contention for locks increase as the number of threads is increased. Figures 13, 16 and 19 show the effect of contention on simulation time when the number of threads is increased. In case of Raid-5 model (Figure 13), we notice the simulation time increases as the number of threads is increased. There is some anomaly in case of 8 threads for Epidemic: Watts-Strogatz (Figure 19) and 16 threads for Epidemic: Fully Connected (Figure 16). This is most likely due to load imbalance on the WARPED scheduler. The simulation times for sorted 2 and 4 LTSF queues steadily increase except for some anomalies in case of (2

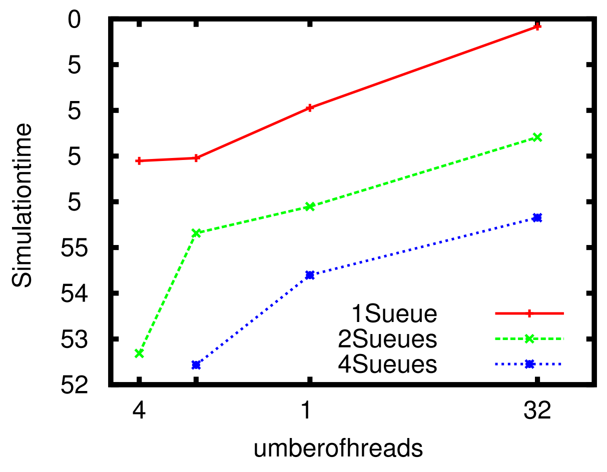


Figure 13: Raid-5 model: Sorted Ladder Queue simulation time

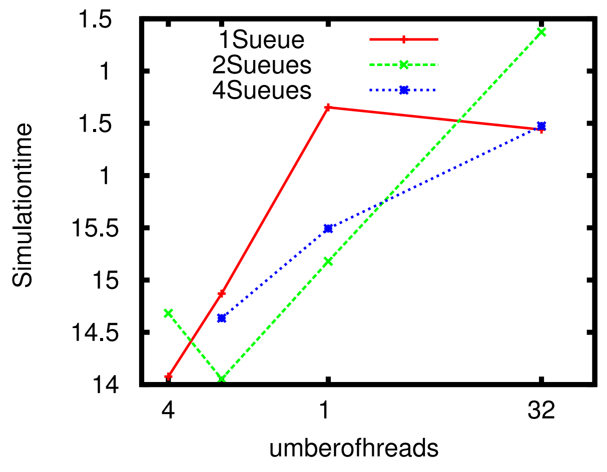


Figure 16: Epidemic model (Fully Connected): Sorted Ladder Queue simulation time

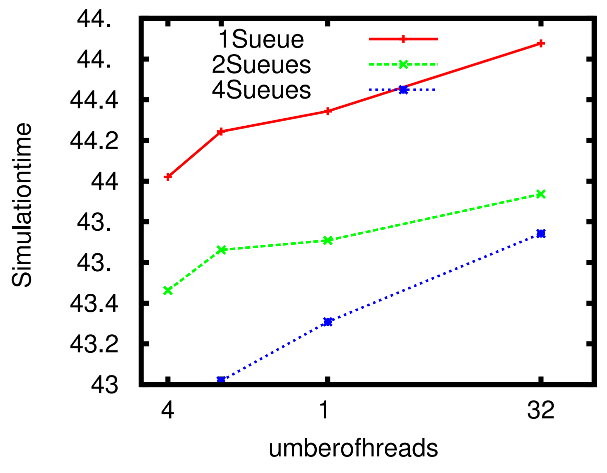


Figure 14: Raid-5 model: Lock-free unsorted Ladder Queue simulation time

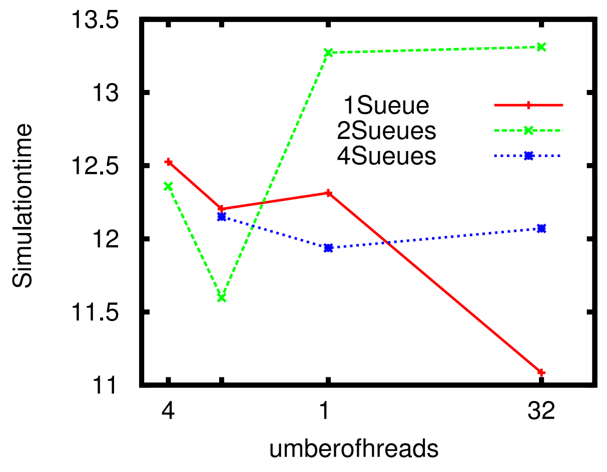


Figure 17: Epidemic model (Fully Connected): Lock-free unsorted Ladder Queue simulation time

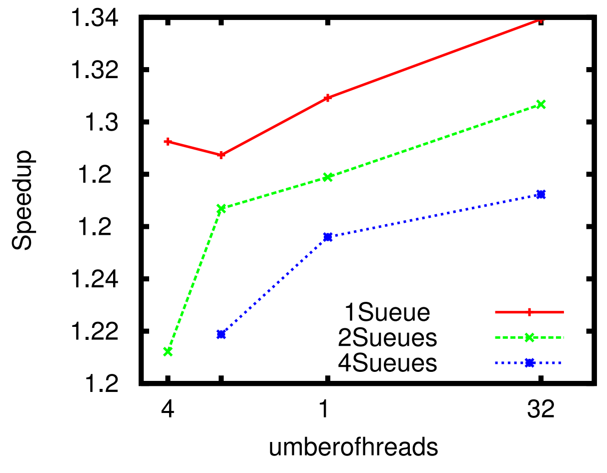


Figure 15: Raid-5 model: Lock-free unsorted Ladder Queue vs. Sorted Ladder Queue.

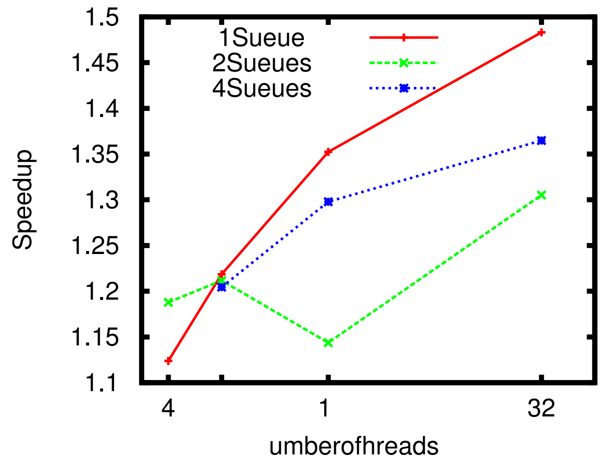


Figure 18: Epidemic model (Fully Connected): Lock-free unsorted Ladder Queue vs. Sorted Ladder Queue.

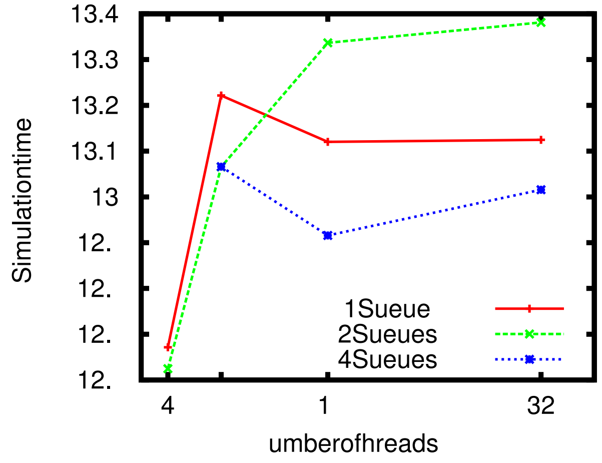


Figure 19: Epidemic model (Watts-Strogatz): Sorted Ladder Queue simulation time

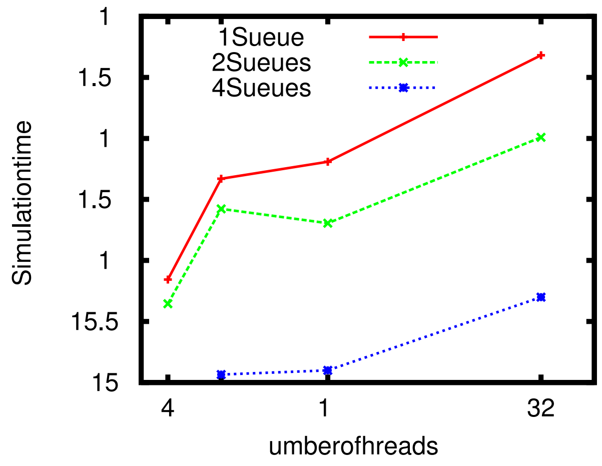


Figure 20: Epidemic model (Watts-Strogatz): Lock-free unsorted Ladder Queue simulation time

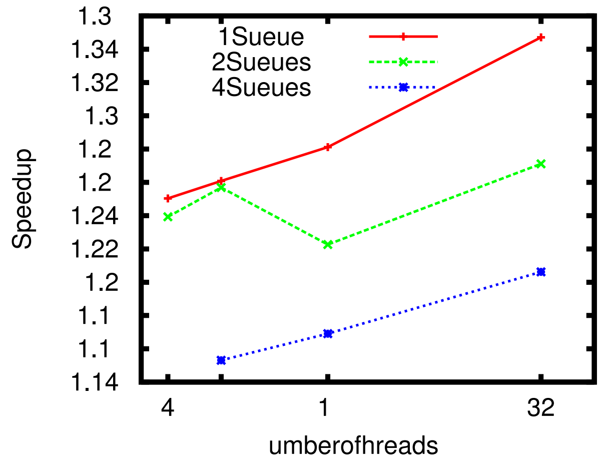


Figure 21: Epidemic model (Watts-Strogatz): Lock-free unsorted Ladder Queue vs. Sorted Ladder Queue.

LTSF, 4 threads) for Epidemic: Fully Connected (Figure 16) and (4 LTSF, 8 threads) for Epidemic: Watts-Strogatz (Figure 19).

On the other hand, a lock-free queue shows relatively steady performance even when the number of threads is increased. Figures 14, 17 and 20 show a relatively steady behavior as the number of threads is increased. However, Epidemic: Fully Connected (Figure 17 shows some anomaly in case of 2 LTSF queues. The Fully Connected model essentially relies on a random network and hence such anomalies are possible. If there are very few causality violations when events are scheduled using Unsorted Ladder Queue, the number of rollbacks will remain relatively low — effectively improving the simulation runtime.

In general, the speedup ratio should increase as the number of threads per LTSF queue increases and vice versa. In case of 4 threads, we notice significant speedup in spite of limited contention overhead for a sorted `Ladder Queue`. This is due to effectiveness of `relaxed causality` in the unsorted `Ladder Queue`. The Raid-5 model (Figure 15) shows up to 29% speedup while the Epidemic: Fully Connected (Figure 18) and Epidemic: Watts-Strogatz (Figure 21) shows up to 20% and 25% speedup respectively for 4 threads. The unsorted `Bottom` in lock-free `Ladder Queue` encourages relaxed causality (refer Section 4). Speedup figures for 8, 16 and 32 threads too are boosted by relaxed causality and show an upward trend except in case of (2 LTSF, 16 threads) for the two epidemic models.

6. DISCUSSION

Based on the results presented in Section 5, it can be said that the `Lock-Free Ladder Queue` with unsorted `Bottom` performs very well. The `Ladder Queue`'s ability to coarsely sort events into groups which are causally independent is a key feature that we have attempted to highlight in this paper. However, this property is dependent on the nature of event pool. If a model generates a considerable amount of causally dependent events within any given epoch, then the benefits of using unsorted `Bottom` would be considerably diminished. A `Sorted Ladder Queue` might yield better results under such circumstances. That said, our models did not show this behavior.

Pienta *et al* [21] analyzed the effect of scale-free model topology on conservative parallel simulation. They found that with increase in degree of any hub node, the hub node needs to process more events during each epoch leading to loss of parallel speedup. The scheduler design we have proposed in this paper should ameliorate the speedup problem for such hub nodes.

The other aspect of this paper is the `lock-free` nature of the WARPED scheduler. While we have migrated to atomic moves for the `Ladder Queue` with unsorted `Bottom`, we have retained the locks for sorted `Ladder Queue`. The `Bottom` structure in sorted `Ladder Queue` uses `STL multiset` to hold events in sorted order. Heller *et al* [8] proposed a lock-based optimistic list implementation with wait-free lookup options. Based on the results presented in [33], this lock-based implementation shows performance superior to the lock-free list implementation used for the `Ladder Queue` with unsorted `Bottom`. Whether or not this lock-based list implementation will boost performance of WARPED can only be answered through experimental analysis in future.

7. CONCLUSIONS

In this paper, we explore the concept of `relaxed causality` using an unsorted `Bottom` in a `Ladder Queue` that is used to manage the pending event sets for a Time Warp parallel simulation engine. We propose the replacement of unsorted lists in `Top`, `Rungs` and `Bottom` with unsorted lock-free queues. Experimental analysis shows a speedup of 1.1 to 1.49 for two different simulation models in various configurations of threads and schedule queues. The results further show that the speedup results are in the higher region of these speedups with larger thread counts. These results help to support our hypothesis that multiple LTSF queues, each made of lock-free coarsely partitioned events in a `ladder queue`, is an efficient candidate for scheduling events in a Time Warp simulator.

The organization and management of the pending event set is critical to the performance of a multi-threaded Time Warp synchronized parallel simulation engine. The two-level pending event set implemented with an unsorted lock-free Ladder Queue provides the best performance among the variety of configurations that we have examined. In a separate and as yet unreported study, we have initiated studies with hardware-based transactional memory to manage the updates to the pending event set. These studies are showing a slight improvement in performance (on the order of 10%) over the conventional mutex locked ladder queue implementation of the LTSF queues. The reason these speedup numbers are not higher is due to the collisions that still occur among the threads to this shared structure. Interestingly enough, the studies with transactional memory have also caused us to identify an approach that should further improve performance for several of our existing solutions. Specifically we now believe that it is best to decouple the worker threads from the LTSF queues and then to rotate each access by a worker thread to the next LTSF queue (in a circular manner).[1] This will spread out the concurrent access into different data segments (each of the different LTSF queues), reducing collisions to transactional data accesses and/or mutex locks/shared data. Almost even more importantly, this will also have the side benefit of distributing the critical path of LPs among the LTSF queues. Thus, load balancing will occur naturally as a result of the event management process rather than as a separate disruptive process (such as the technique outlined in [4]).

8. ACKNOWLEDGMENTS

Support for this work was provided in part by the National Science Foundation under grant CNS–0915337.

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] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe. Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 37:1–37:12, 2008.

[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] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for pdes on many-core beowulf clusters. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 103–114, May 2013.

[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] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314. Springer-Verlag, 2001.

[8] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, 2006.

[9] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[10] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, 2011.

[11] J. Liu, D. M. Nicol, and K. Tan. Lock-free scheduling of logical processes in parallel simulation. In *Proceedings of the Fifteenth Workshop on Parallel and Distributed Simulation*, PADS '01, pages 22–31. IEEE Computer Society, 2001.

[12] 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.

[13] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, 2002.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[15] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, Rochester, NY, USA, 1995.

[16] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.

[1]This will introduce yet another location of contention for the worker threads. However it should be nothing more than a fetch-and-add type operation by each worker thread to retrieve the next index into the pool of LTSF queues.

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

[18] 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.

[19] 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.

[20] K. S. Perumalla and S. K. Seal. Discrete event modeling and massively parallel execution of epidemic outbreak phenomena. *Simulation*, 88(7):768–783, July 2012.

[21] R. S. Pienta and R. M. Fujimoto. On the parallel simulation of scale-free networks. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 179–188, 2013.

[22] 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.

[23] R. Radhakrishnan, D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. An Object-Oriented Time Warp Simulation Kernel. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume LNCS 1505, pages 13–23. Springer-Verlag, Dec. 1998.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 309–310, 2012.

[31] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, 1995.

[32] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

[33] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear. Practical non-blocking unordered lists. In *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 239–253. Springer Berlin Heidelberg, 2013.