Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

OPTIMIZING EVENT TIMESTAMP PROCESSING IN TIME WARP

Gaurav Shinde[1], Sounak Gupta[2], and Philip A. Wilsey[3]

[1]STERIS, Mentor, OH, USA
[2]Oracle, San Jose, CA, USA
[3]Dept. of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, OH, USA

ABSTRACT

WARPED2 is a discrete event simulation kernel that contains a robust event time comparison mechanism to support a broad range of modeling domains. The WARPED2 kernel can be configured for sequential, parallel, or distributed execution. The parallel or distributed versions implement the Time Warp mechanism (with its rollback and relaxed causality) such that a total order on events can be maintained. To maintain a total order, WARPED2 has an event ordering mechanism that contains up to 10 comparison operations. While not all comparisons require evaluation of all 10 relations, the overall cost of time comparisons in a WARPED2 simulation can still consume approximately 15-20% of the total runtime. This work examines the runtime costs of time comparisons in a parallel configuration of the WARPED2 simulation kernel. Optimizations to the time comparison mechanism are explored and the performance impacts of each are reported.

1 INTRODUCTION

The Time Warp mechanism is an optimistic synchronization strategy for *Parallel and Distributed Discrete-Event Simulation (PDES)*. The mechanism organizes a parallel or distributed simulation into a collection of concurrently executing and fully asynchronous discrete-event simulations called *Logical Processes (LPs)*. These LPs operate concurrently on their local discrete event lists without an explicit synchronization infrastructure coordinating the LPs. Events generated by one LP for another LP are communicated with a timestamp and if they arrive in the simulation past of the receiving LP a *rollback* will occur in the receiving LP to re-establish the causal order of event processing by the LP. During a rollback, it may be necessary to cancel any prematurely sent events; this is achieved by sending *anti-messages* to remove the premature events from the destination LP's event space. Thus, the concurrency of LP event processing, rollback, and event cancellation thorough anti-messages can generate events and anti-messages with equivalent timestamps and the challenge to achieve a total order for LP event processing can require additional information beyond the simple simulation time that the event is to be processed. Note, it is not necessary to have a strict total order of event processing of different LPs; the optimistic model does not require it. However, within an LP, many simulation models will require a strict total order on the events processed by that LP.

In a Time Warp PDES, achieving a total order on events is complicated by several factors. For example, it is possible that a set of parallel LPs can generate events with the same timestamp destined for the same LP; furthermore, any or all of the sending LPs can possibly rollback, terminating the prematurely sent event (using an *anti-message*), and ultimately re-transmitting another event (potentially identical to the original) with the same timestamp to the same destination LP (Fujimoto 1990). In this case, the destination LP must ensure that all events are processed following a total order that preserves causality relationships (Fujimoto 2000). While not all simulation models require a total order on all events, many do. Thus, a general purpose Time Warp simulation engine must be capable of organizing a total order on events.

This paper examines the runtime performance costs of event comparison in the parallel and distributed configuration of the WARPED2 simulation kernel that implements the Time Warp mechanism. In particular, profiling shows that as much as 21% of the runtime costs is consumed by the timestamp comparison operator

in an execution of the WARPED2 simulation kernel. This study will examine the performance of 4 different configurations for the event queue, namely: (i) implemented with as a fully sorted `std:multiset` (from the `c++` standard library), (ii) as a Ladder Queue (Tang, Goh, and Thng 2005; Dickman, Gupta, and Wilsey 2013) (fully sorted), (iii) as a Ladder Queue (partially sorted) (Gupta 2018), and (iv) as a Unified Queue with a relaxed timestamp comparison mechanism. In addition to studying and optimizing the timestamp comparison function, the various uses of the timestamp comparison in WARPED2 is examined to determine if and when the costly total order comparison operation can be replaced with a more relaxed (and faster) comparsion operator. Finally, this paper will discuss which data fields are absolutely necessary and which can potentially be eliminated; various mechanisms to reduce the cost of timestamp comparison are implemented and studied in scenarios where total ordering is not strictly necessary.

Among the configurations tested, the Unified Queue implementation demonstrates the most substantial improvements, reducing event comparison costs by up to 76% in complex models compared to the Multiset implementation. However, based on the model used for benchmarks, when rollback frequency increases there is a heavy performance penalty with the Unified Queue implementation. The partially sorted Ladder Queue shows promise as a middle ground option. This paper also explores Hash Event Set and Single-Instruction Multiple-Data (SIMD) implementations; however the latter failed to deliver significant performance gains on AVX2 platforms. The research concludes with recommendations for architectural refinements, hybrid queue structures, memory optimizations, and advanced SIMD implementations to further improve performance.

The remainder of this paper is organized as follows. Section 2 contains some background and related work. It also presents and explains the computations of the `compareEvent` operation that establishes a total order of events WARPED2. Section 3 presents a new, lightweight `compareEvent` operation and a new input queue (Unified Queue) that combines processed and unprocessed events for WARPED2. Section 4 uses profiling results to examine the runtime costs of timestamp comparisons in the WARPED2 simulation kernel. Section 5 examines the memory usage characteristics captured from executions of simulations models running on WARPED2. Section 6 examines strategies to reduce the runtime costs of the timestamp comparison operation. Section 7 summarizes the results and lessons learned from the profiling based analysis and performance comparison of this paper. Finally, Section 8 contains some concluding remarks.

2 BACKGROUND AND RELATED WORK

Lamport's seminal work on distributed systems established the foundational theory for understanding event ordering across multiple processes (Lamport 1978). By formalizing the *happened-before* relation, Lamport provided a mathematical framework for capturing the partial ordering of events that naturally emerges in distributed environments. To represent this ordering computationally, he introduced logical clocks that assign timestamps to events with the critical property that if event `a` causally precedes event `b`, then the timestamp `C(a)` must be less than `C(b)`. Recognizing that causality alone only yields a partial ordering, Lamport extended this to a total ordering through a simple but elegant mechanism: using process identifiers to break ties when events have identical timestamps. This innovation proved crucial for deterministic event processing in distributed simulations. Perhaps most importantly, Lamport demonstrated how reliance on physical time could lead to counter-intuitive and seemingly impossible behaviors in distributed systems, thus establishing the theoretical justification for logical time—a concept that would later become central to parallel discrete event simulation techniques.

Jefferson's Time Warp protocol built upon Lamport's foundation of logical time but introduced the optimistic synchronization approach with *rollbacks* and *anti-messages* (Jefferson 1985). He introduced LPs that process events optimistically without waiting for any causally related events; letting violations happen if they occurred. Violations are corrected after detection rather than prevented. This results in higher parallelism through speculation. When causality violations occur (by a *straggler* event message arriving with a timestamp in an LP's virtual past), the Time Warp employs a rollback mechanism that returns the process to an earlier state where the straggler event can be processed in its causal order. The protocol's innovation lies in its anti-message system—negative copies of previously sent messages that

propagate through the system to cancel the effects of rolled-back computations. This elegant solution enables simulations to maintain correctness while exploiting parallelism, effectively trading occasional rollback overhead for increased throughput, particularly in scenarios where causality violations are infrequent.

In 2016, the WARPED2 PDES simulation kernel implementing the Time Warp Mechanism was released. In this version of WARPED2, each hardware compute node contains a set of *worker threads* and a single *manager (or housekeeping) thread* (in WARPED2, a cluster is a collection of compute nodes and a multi-core processor is considered a single compute node). The *worker threads* serve as the computational engines for the LPs that advance the simulation while the *manager threads* are responsible for *Global Virtual Time (GVT)* calculation, initialization and termination of the program. Multiple worker threads operate simultaneously, with each thread managing events for one or more LPs. The Time Warp mechanism serves as the conductor of this distributed simulation implementation, coordinating all of the moving parts.

Events in WARPED2 follow a life-cycle: they begin as *unprocessed*, transition to *processed*, and finally become *fossilized* (sometimes, oscillating between processed and unprocessed). Throughout this progression, events move through several queue structures: the *Input Queue* (subdivided into processed and unprocessed sections), the *schedule queue*, and the *output queue*. Events are ultimately removed during fossil collection. WARPED2 implements these queues using *RedBlack (RB)* plus trees that require rebalancing whenever events are added or removed. To ensure memory safety, mutex locks are applied for all queue operations. This makes the rebalancing and sorting operations particularly critical: the faster these operations complete, the sooner locks can be released, reducing wait times and accelerating execution.

Event comparison in distributed simulation systems such as WARPED2 requires careful exploration due to several factors that directly impact simulation correctness and performance. The necessity stems from both the fundamental architecture of the system and potential edge cases that must be addressed. Since locks create potential bottlenecks, the comparison operations must be both accurate and efficient; any unnecessary complexity in comparison logic directly translates to longer lock holding times and reduced simulation performance. The distributed architecture of the simulation also creates several intricate challenges that demand careful handling. When simulations run for extended periods, vector time can wrap around, potentially creating confusion in timestamp comparisons. Additionally, different LPs may generate events with matching timestamps, necessitating extra comparison criteria to ensure events are processed in a consistent, predictable order. The system must also maintain the ability to differentiate between standard events, anti-events (during rollback procedures), and any events re-transmitted after a rollback.

2.1 Causal Order in PDES

Developing a precise definition of order in an optimistically synchronized PDES simulation kernel is more challenging than simply order objects based on the simulation timestamp. As a simple example, assume that two concurrently executing LPs (LP_i and LP_j) are generating events (e_i and e_j) with timestamp t to a receive LP (LP_k). Without additional discriminating rules, it is impossible to define a total order on the two events e_i and e_j. This section will explore the problem of establishing causal order in a Time Warp synchronized PDES kernel. This discussion will treat the problem of ordering events in a Time Warp system without addressing optimizing and simplifying concepts such as direct cancellation (Fujimoto 1990) or reverse computation (Carothers, Perumalla, and Fujimoto 1999); instead this discussion will focus on the generic concept of a Time Warp system: copy state saving, and aggressive cancellation (Fujimoto 1990).

In addition to the problem of events generated with a common timestamp, the optimistic processing of events also requires anti-message events to kill a prematurely generated event. In general these are sent with information to match the original premature message including its timestamp. Furthermore, with the optimistic processing and rollback nature of Time Warp, it is possible that an LP will generate and cancel an event at timestamp t multiple times. Any solution to generate a total order on events and anti-message events must account for (i) the possible premature generation, (ii) the possible rollback, and (iii) the possible re-generation of events (at the same timestamp t). The remainder of this paper will examine the solution to this problem as it has been addressed in the WARPED2 simulation kernel (Weber 2016).

Listing 1: Strict comparison functor for events.

```
struct compareEvents {
public:
bool operator() (const std::shared_ptr<Event>& first,
const std::shared_ptr<Event>& second) const {
  return  (first->timestamp() < second->timestamp()) ? true :
          ((first->timestamp() != second->timestamp()) ? false :
          ((first->send_time_ < second->send_time_) ? true :
          ((first->send_time_ != second->send_time_) ? false :
          ((first->sender_name_ < second->sender_name_) ? true :
          ((first->sender_name_ != second->sender_name_) ? false :
          ((first->generation_ < second->generation_) ? true :
          ((first->generation_ != second->generation_) ? false :
          ((first->event_type_ < second->event_type_) ? true :
          ((first->event_type_ != second->event_type_) ? false : false )))))))));
}};
```

The WARPED2 simulation kernel is written in C++ and defines a `compareEvents` operator to establish a total order of events (Listing 1). The fields in the Event class are carefully designed to support the optimistic execution model with a total order on events. Each field serves a specific purpose, from routing events to the correct LPs to handling rollbacks and ensuring the correct order of event processing. Together, these fields enable the simulation to handle complex scenarios involving straggler events, anti-messages, and event regeneration, ensuring the consistency and accuracy of the simulation. In WARPED2, the `compareEvents` tests are short circuited to return a result as quickly as possible. The hierarchy of tests to order events is:

1. The `timestamp()` method returns when an event should be processed, serving as the primary ordering criterion and enabling the detection of out-of-order events that trigger rollbacks. Events with earlier timestamps are processed first, maintaining chronological progression.
2. The `send_time` field captures when an event was created, functioning as a secondary discriminator when events share identical timestamps. This additional precision helps preserve processing sequence even when multiple events occur simultaneously.
3. Each event's `sender_name` identifies its originating Logical Process, providing a third-level distinction particularly valuable in distributed environments. This field ensures consistent event ordering across multiple sources even when timestamps and send times match.
4. For managing complex rollback scenarios, the `generation_field` distinguishes between otherwise identical events regenerated after system rollbacks. When straggler events force the simulation to revert, previously canceled events will be regenerated with incremented generation values.
5. The `event_type` field completes the ordering system by differentiating between regular events and anti-messages. When rollbacks occur, these negative events cancel the effects of previously processed messages, allowing the simulation to maintain consistency while proceeding optimistically.

Collectively, these tests create a deterministic total ordering. The system reliably processes events in the correct sequence while supporting the optimistic execution model that makes Time Warp simulations both efficient and accurate; even across distributed environments with complex causal relationships.

3 UNIFIED QUEUE & RELAXED COMPARE EVENTS: UNIFIED QUEUE

All of the current implementations of the input queue in WARPED2 separate the input queue into two queues: one for *unprocessed events* and one for *processed events*. Both queues are sorted and events move from the unprocessed to the processed queue (and back) for forward event processing (and on rollback). This paper presents a "Unified Queue" input queue that works to reduce the number of calls (and thus its cost) to the

`compareEvent` operator. In addition, this work will also define and use a `relaxedCompareEvent` operator (Listing 2) that sorts events using only the `timestamp()` method (Shinde 2024). The use of `relaxedCompareEvent` in the WARPED2 queue structures is explained more fully below.

Listing 2: Relaxed Comparison functor for events.

```
struct relaxedCompareEvents {
public:
bool operator() (const std::shared_ptr<Event>& first,
const std::shared_ptr<Event>& second) const {
        return (first->timestamp() < second->timestamp());
}};
```

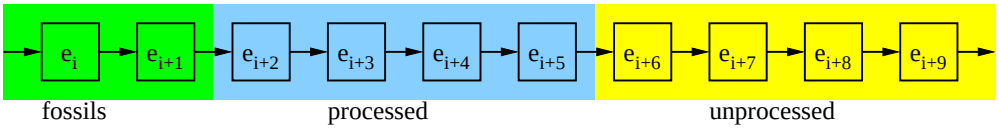

Figure 1: Unified queue as the LP input queue.

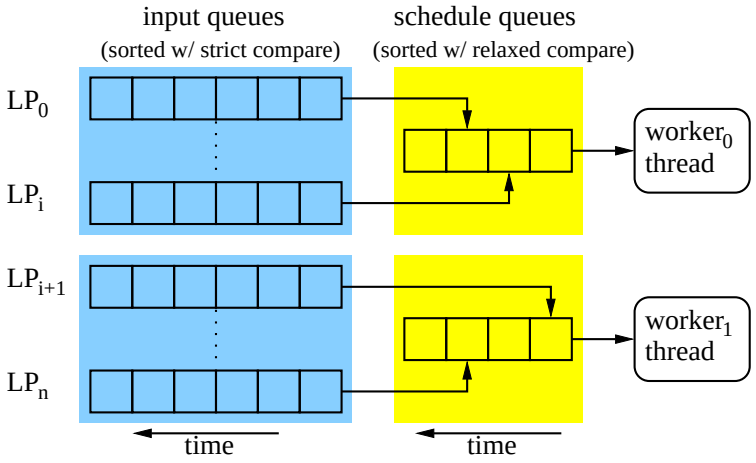

Figure 2: Comparisons/Ordering in WARPED2 v2.x.

The Unified Queue combines the processed and unprocessed events into a single circular queue (Figure 1). Used for the input queue, the Unified Queue is composed of 3 zones, namely: (i) unprocessed events (the yellow shading), (ii) processed events yet to be fossilized (light blue shading), and (iii) fossilized events (light green). This organization and a careful construction of GVT records permits concurrent access in event processing and fossil collection such that locking between the event processing threads and the fossil collection threads is not required (Shinde 2024).

As previously indicated each compute node in a WARPED2 PDES configuration is organized as a collection of *worker* threads and *housekeeping* threads. The housekeeping threads address fossil collection, GVT management, and remote message communications (in a distributed configuration). While some queue access is performed by the housekeeping threads, the majority of the queue accesses are made by the worker threads. In general, the best performance of a WARPED2 PDES model is to define one worker thread for each hardware thread with a static partition of LPs to each worker thread (Figure 2). To optimize input queue processing, the WARPED2 system defines a schedule queue in front of the input queues of the LPs assigned to each worker threads. The schedule queue is populated by the lowest timestamped event

from the input queue of each LP. After an event is removed from the schedule queue and processed by the worker thread the next timestamped event from the corresponding LP is added to the schedule queue.

Prior to this work, the WARPED2 schedule queue has always been sorted by the strict `compareEvents` operator from Listing 1. This paper will, however, consider using the `relaxedCompareEvent` operator to sort events in the schedule queue. The remainder of this paper will examine profiling and performance results from various implementations of the input queue with the strict and relaxed comparison operators.

4 PROFILING THE COST OF EVENT COMPARISONS

The results are presented using 3 different simulation models that possess a broad range of event processing granularities and coupling of events among the LPs. The 3 models are: (i) an epidemic disease propagation model (Epidemic) based on (Perumalla and Seal 2012), (ii) a Portable Cellular Service (PCS) network simulation, and (iii) an urban traffic model (Traffic) adapted from ROSS (Carothers, Bauer, and Pearce 2000). The epidemic model employs *Probabilistic Timed Transition System (PTTS)* (Barrett, Bisset, Eubank, Feng, and Marathe 2008) for disease progression and implements both Watts-Strogatz (Watts and Strogatz 1998) and Barabási-Albert (Barabási and Albert 1999) networks for modeling disease spread between locations. The PCS model simulates cellular network operations including call handling and device migration. The traffic model simulates vehicle movement through a city grid, with each intersection as a logical process (LP). The experimental evaluation examines these models across four configurations, varying the number of LPs (10k-100k) and worker threads (3-6), with consistent sample sizes and GVT periods.

Table 1: VTune top 5 hotspots for WARPED2 configurations Traffic-20k.

Function	Multiset	Fully Sorted Ladder Queue	Partially Sorted Ladder Queue	Unified Queue
			Seconds	
compareEvents::operator()	23.39	15.75	14.771	12.4 (relaxed+strict)
operator new	7.5	8.089	8.62	–
chrono::_V2::steady_clock::now	7.41	5.64	6.60	–
___pthread_mutex_lock	5.6	5.23	5.01	7.910
map<string, uint>::operator[]	5.3	–	–	–
__GI___libc_mallinfo2	–	5.62	5.7	11.171

Table 2: VTune top 5 hotspots for WARPED2 configurations PCS-5k.

Function	Multiset	Fully Sorted Ladder Queue	Partially Sorted Ladder Queue	Unified Queue
			Seconds	
compareEvents::operator()	137.228	115.59	72.184	43.101 (relax + strict)
operator new	27.197	30.159	25.071	–
chrono::_V2::steady_clock::now	34.742	32.703	24.947	57.287
___pthread_mutex_lock	27.97	24.027	21.707	45.974
map<string, uint>::operator[]	5.3	–	–	–
__GI___libc_mallinfo2	32.730	39.660	28.110	–
UnifiedQueue::binarySearch()	–	–	–	35.358
shared_ptr<Event> operator =	–	–	–	101.756

Tables 1, and 2 present performance profiling data from a VTune™ analysis across 4 different queue implementations in WARPED2: (i) Multiset, (ii) Ladder Queue (fully sorted), (iii) Ladder Queue (partially sorted), and (iv) Unified Queue (Gupta 2018; Shinde 2024). The analysis was performed using 2 simulation models, namely: Traffic-20k and PCS-5k.

Event comparison (`compareEvents::operator()`) consistently emerges as the primary computational bottleneck across all implementations. In the Traffic-20k model, the Unified Queue demonstrates superior performance, reducing comparison time to 12.4 seconds: a 47% improvement over the Multiset

implementation (23.39 seconds). Both Ladder Queue variants show intermediate improvements, with the partially sorted variant (14.77 seconds) outperforming the Fully Sorted variant (15.75 seconds). This pattern becomes more pronounced in the computationally intensive PCS-5k model, where the Unified Queue reduces comparison time to 43.10 seconds, compared to 137.23 seconds with the Multiset implementation—a dramatic 68.6% reduction. The partially sorted Ladder Queue (72.18 seconds) delivers significantly better performance than the fully sorted variant (115.59 seconds), suggesting that partial sorting offers a meaningful efficiency advantage in more complex simulations.

Memory allocation costs (operator `new`) remain relatively consistent across implementations in both models, ranging from 7.5 to 8.62 seconds in Traffic-20k and 25.07 to 30.16 seconds in PCS-5k. The Unified Queue implementation eliminates this specific overhead from its top hotspots, but introduces new memory-related costs with significant overhead from shared pointer operations (101.76 seconds) in the PCS-5k model. Memory tracking operations (`__GI___libc_mallinfo2`) show variation between implementations, with higher costs in the Unified Queue for Traffic-20k (11.17 seconds) compared to the Ladder Queue variants (approximately 5.7 seconds). This overhead does not appear among top hotspots for the Unified Queue in the PCS-5k model.

Mutex operations (`___pthread_mutex_lock`) consume between 5.01 and 7.91 seconds in Traffic-20k, with the Unified Queue showing the highest synchronization cost. This pattern is amplified in PCS-5k, where mutex operations in the Unified Queue (45.97 seconds) are approximately twice as expensive as in the Ladder Queue implementations (21.71 to 24.03 seconds). Timing operations (`chrono::_V2::steady_clock::now`) demonstrate similar variation, with the Unified Queue showing lower overhead in Traffic-20k but substantially higher costs in PCS-5k (57.29 seconds compared to 24.95-34.74 seconds).

The Unified Queue introduces unique operational costs, including binary search operations (35.36 seconds in PCS-5k) and shared pointer assignment operations (101.76 seconds in PCS-5k), which do not appear as significant hotspots in other implementations. The Multiset implementation shows additional overhead from map operations (`map<string, uint>::operator[]`), consuming 5.3 seconds in both models, a cost not present in the other implementations' top hotspots.

Performance characteristics vary significantly between the Traffic-20k and PCS-5k models. While relative performance trends remain consistent, the PCS-5k model amplifies computational costs across all operations. This suggests that implementation efficiency becomes increasingly critical as model complexity increases, with the most efficient implementations showing the greatest relative improvement in more demanding scenarios. Overall, these results indicate that optimizing event comparison operations provides the most substantial performance improvements in discrete event simulations, with the Unified Queue and partially sorted Ladder Queue offering the most efficient alternatives to traditional Multiset implementations.

5 MEMORY ANALYSIS

Table 3, lists the metrics generated by Intel VTune™ Memory Benchmark. WARPED2 spends 30% of the CPU's execution time waiting for memory operations to complete. This is a significant bottleneck, as the CPU is frequently stalled due to slow memory access. In this case, the primary contributors to Memory Bound are L1 Bound (15.0%) and DRAM Bound (23.8%), which together account for most of the 29.5% Memory Bottleneck. Other factors such as L2, L3, Store do not contribute much to this metric. The Loads metric shows $50,322,246,627$ load operations, but the LLC Miss Count (Last Level Cache misses) is $148,914,276$. This means that approximately 0.3% of load operations result in LLC misses. LLC Miss Counts means that the value could not be resolved by L1, L2, and L3 caches and the value was fetched from DRAM. The application performs a large number of load and store operations, indicating that it is memory-intensive. The high number of loads relative to stores suggests that the WARPED2 is reading data more frequently than it is writing. Reducing the number of memory accesses, particularly loads, can improve performance. This could be achieved by reusing data in registers or cache, optimizing algorithms to minimize memory traffic, or using more efficient data structures.

Table 3: VTune™ memory usage metrics Epidemic.

Metric	Value
Elapsed Time	19.102 s
CPU Time	58.709 s
Memory Bound	29.5%
L1 Bound	15.0%
L2 Bound	1.8%
L3 Bound	4.0%
DRAM Bound	23.8%
Store Bound	0.1%
Loads	50,322,246,627
Stores	25,499,413,254
LLC Miss Count	148,914,276
Total Thread Count	10
Paused Time	0 s

Table 4: VTune top 5 hotspots for WARPED2 configurations Traffic-20k.

Function	RB WARPED2	`xxHash` Event
	Seconds	
Elapsed Time	34.951	33.750
CPU Time	116.480	116.870
Paused Time	0	0
compareEvents::operator()	23.397	18.416
operator new	7.490	–
chrono::_V2::steady_clock::now	7.411	7.640
___pthread_mutex_lock	5.679	5.649
map<string, uint>::operator[]	5.3	–
__GI___libc_mallinfo2	5.530	5.320
std::unordered_map<>[]	–	5.058

6 STRATEGIES TO REDUCE COMPARISON COSTS

6.1 Hash Event Set

The original `compareEvent` function performs comparisons on `sender_name` which is a string in WARPED2. Of course, string comparisons are more expensive than numeric comparisons. As a result, a hashing mechanism that shields model designers from kernel-level implementation details while reducing comparison operator costs was implemented. The `xxHash` library is used to generate the hash of each LP.

VTune™ profiling data for the Traffic-20k simulation model using (i) the original comparison code and (ii) the `xxHash`-based approach are shown in Table 4. Both configurations demonstrate similar overall execution metrics with elapsed times of approximately 34.95 seconds and 33.75 seconds respectively. The CPU time is nearly identical (116.48 vs 116.87 seconds), suggesting effective parallel execution across multiple threads. The CPU time being more than three times the elapsed time confirms substantial multi-threading benefits in both configurations. The event comparison operation (`compareEvents::operator()`) emerges as the primary performance bottleneck, consuming approximately 20% of the CPU time in the original code and 16% in the `xxHash` version. This 5% reduction suggests that the `xxHash`'s event comparison approach provides a modest performance advantage. Memory management operations appear as significant hotspots in both configurations. The original code spends notable time in memory allocation (operator `new` at 7.49 seconds), while both implementations spend similar amounts of time in `chrono::_V2::steady_clock::now` (7.41 and 7.64 seconds respectively) and `mutex` locking operations (approximately 5.65 seconds). These findings indicate that, while the `xxHash` implementation provides a modest overall performance improvement (3.5% faster elapsed time), the primary performance constraints remain consistent across both implementations: (i) event comparison operations, (ii) memory management, and (iii) synchronization overhead. Future optimization efforts should focus on these common bottlenecks, particularly exploring ways to reduce comparison costs and mutex contention in the discrete event simulation engine.

6.2 SIMD

The possibility of exploiting SIMD operations was also evaluated in the optimization of the `compareEvent` operator. To perform this study, the `sender_name` string and the WARPED2 `compareEvent` operator is converted to the form shown in Listing 3. The data presented in Figures 3 and 4, compares the performance of two different computational approaches, SIMD (Single Instruction, Multiple Data) and RB, across three distinct models: "epidemic 10k", "PCS 1k", and "traffic 20k". The comparison is made on two key metrics: (i) total simulation runtime and (ii) the runtime per event. In general the results show that while the SIMD

approach may be more efficient for larger models such as "epidemic 10k", the advantage shrinks for models with smaller execution (per event) costs. These tests were performed on machines supporting the AVX2 SIMD extensions; results with AVX-512 SIMD extensions may show improved results.

Listing 3: SIMD implementation for event comparison.

```
class Event {
public:
// ...  other members as before ...

void generateHash(){
        senderHashId_ = XXH64(sender_name_.data(), sender_name_.size(), 0); // Seed = 0
        // Assign data into array for SIMD comparison
        data_[0] = timestamp();
        data_[1] = send_time_;
        data_[2] = senderHashId_;
        data_[3] = generation_;  }

// SIMD-based comparison operators
bool operator== (const Event &other) const {  }
bool operator< (const Event &other) const {  }

static constexpr size_t EVENT_DATA_SIZE = 4;
alignas(32) std::array<uint64_t, EVENT_DATA_SIZE> data_; // 32-byte aligned for AVX2
};
```

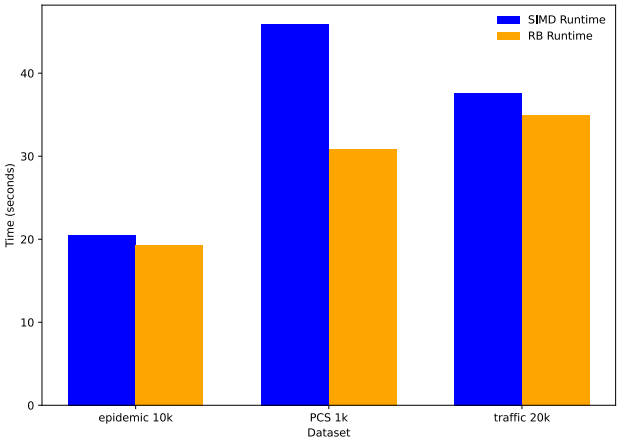

Figure 3: Comparison of SIMD and RB runtimes.

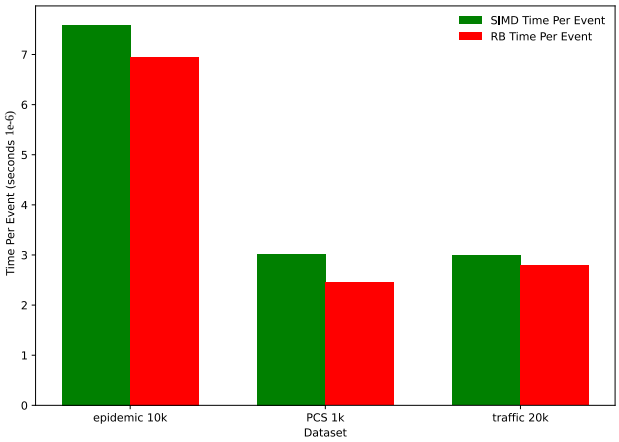

Figure 4: SIMD and RB time per event.

7 DISCUSSION

The performance analysis conducted throughout this study reveals several critical insights into the optimization of parallel discrete event simulations. Event comparison emerges as the predominant computational bottleneck across all WARPED2 implementations, consuming a significant portion of CPU time—ranging from 12.4 seconds in the optimized Unified Queue to 23.39 seconds in the traditional Multiset implementation for the Traffic-20k model (Table 1), and from 33.10 to 137.23 seconds respectively for the more complex PCS-5k model (Table 2). These findings underscore the central importance of efficient event comparison mechanisms in PDES systems.

Table 5: Latency Comparison: Event Class vs Hashed Event Class.

Event Class	**99%**	**90%**	**50%**
Old Event Class	408ns	218ns	94ns
Hashed Event Class	370ns	216ns	88ns

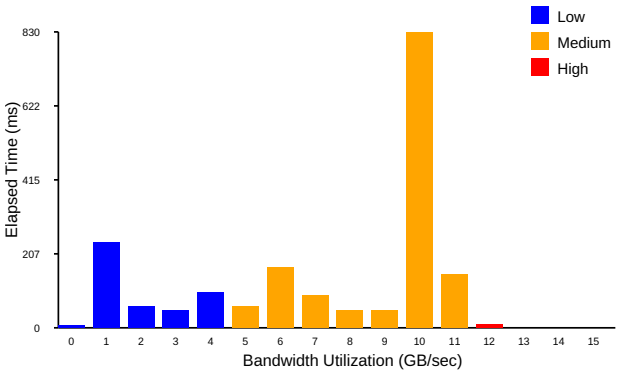

Figure 5: Memory bandwidth utilization traffic.

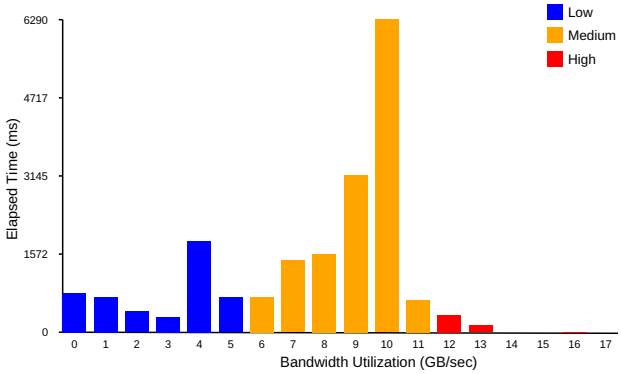

Figure 6: Memory bandwidth utilization epidemic.

Among the approaches evaluated, the Unified Queue implementation demonstrates the most substantial improvements in event comparison costs, achieving up to a 76% reduction in the PCS-5k model compared to the Multiset implementation. This dramatic improvement can be attributed to the queue's architectural advantages that enable the use of relaxed comparison operators for the schedule queue. However, the Unified Queue introduces new computational costs, particularly in shared pointer operations and mutex synchronization, which become increasingly significant in more complex models.

The partially sorted Ladder Queue offers a promising middle ground, with event comparison costs reduced by 47% in PCS-5k compared to the Multiset, without incurring the same level of synchronization overhead as the Unified Queue. This suggests that implementing partial sorting strategies may provide substantial benefits without requiring the architectural overhaul necessitated by the Unified Queue approach.

The Hash Event Set strategy shows moderate performance improvements in comparison latency, with reductions of up to 9.3% for the 99^{th} percentile case (Table 5). While not as dramatic as the queue restructuring approaches, this strategy requires minimal architectural changes and could be implemented as a complementary optimization alongside other techniques.

Contrary to expectations, the SIMD implementation failed to deliver significant performance improvements across the tested models, as shown in Figures 3 and 4. In fact, the RB approach consistently outperformed SIMD in terms of total runtime, particularly for smaller models. This counter-intuitive result suggests that the current SIMD implementation on AVX2 platforms may not be optimally aligned with WARPED2's computational patterns, or that the overhead of setting up SIMD operations outweighs their benefits for the specific comparison operations in WARPED2.

The memory analysis illuminates critical challenges facing WARPED2's performance. With approximately 30% of CPU execution time spent waiting for memory operations, and significant L1 (15.0%) and DRAM (23.8%) bound metrics (Table 3), it is evident that memory access patterns play a crucial role in overall simulation efficiency. The high load operations count (50,322,246,627) compared to store operations (25,499,413,254) indicates a read-heavy computational pattern, which could benefit from strategies that improve data locality and reduce cache misses 3.

Bandwidth utilization analysis reveals that memory allocation functions (`int_mallinfo`, `_int_malloc`) and tree operations (`std::_Rb_tree_increment`) are significant contributors to LLC misses. The memory bandwidth utilization patterns in Figures 5 and 6 further illustrate these memory access challenges

across different simulation models. This suggests that strategies focusing on reducing heap allocations in favor of stack usage, and implementing more cache-efficient data structures for event queues, could yield substantial performance improvements.

8 CONCLUSIONS

This work demonstrates the critical importance of comprehensive profiling methodologies in identifying and addressing performance bottlenecks in complex parallel discrete event simulation systems. Our systematic approach to performance analysis, utilizing multiple profiling tools and metrics, provided insights that would not have been achievable through traditional timing-based performance measurements alone.

The multi-layered profiling approach employed in this study—combining execution time analysis, memory hierarchy profiling, and bandwidth utilization measurements—proved highly effective in uncovering the root causes of performance limitations. The discovery that event timestamp comparison operations consume up to 21% of total runtime was only possible through detailed CPU profiling that went beyond simple wall-clock measurements. This level of granularity enabled us to target specific computational hotspots rather than pursuing broad, potentially ineffective optimizations.

The memory hierarchy analysis was particularly valuable, revealing that 30% of CPU execution time was spent waiting for memory operations. Without detailed cache miss analysis and memory bandwidth profiling, we might have focused exclusively on algorithmic improvements while overlooking the substantial impact of memory access patterns. The identification of specific functions (`int_mallinfo`, `_int_malloc`, `std::_Rb_tree_increment`) as major contributors to LLC misses provided actionable optimization targets.

Understanding the behavior at different levels of the memory hierarchy proved essential for effective optimization strategy development. The L1 cache bound metrics (15.0%) and DRAM bound metrics (23.8%) provided a clear picture of where the memory bottlenecks occurred, guiding our optimization efforts toward data locality improvements rather than strictly on algorithmic changes. Furthermore, the high load operations count ($50,322,246,627$) compared to store operations ($25,499,413,254$) indicates a read-heavy computational pattern that could benefit from strategies to improve data locality and reduce cache misses.

The detailed profiling results enabled a targeted optimization approach that addressed specific bottlenecks rather than attempting broad system-wide changes. The success of the Hash Event Set implementation—achieving 9.3% latency improvement with minimal architectural changes—exemplifies how profiling-guided optimization can deliver meaningful results with focused effort.

Conversely, the profiling methodology also prevented wasted effort on ineffective optimizations. The SIMD implementation results, which showed no significant improvement despite theoretical advantages, were clearly interpretable through our detailed analysis. The profiling data suggested that setup overhead and instruction pipeline characteristics, rather than raw computational throughput, were the limiting factors for SIMD effectiveness in this context.

This study demonstrates that effective performance optimization in complex systems requires profiling methodologies that capture multiple performance dimensions simultaneously. Traditional approaches focusing solely on execution time would have missed the critical memory access patterns that ultimately limit system performance. The combination of temporal profiling (execution time), spatial profiling (memory access patterns), and architectural profiling (cache behavior) provided a comprehensive view necessary for informed optimization decisions.

REFERENCES

Barabási, A.-L., and R. Albert. 1999. "Emergence of Scaling in Random Networks". *Science* 286(5439):509–512 https://doi.org/10.1126/science.286.5439.509.

Barrett, C. L., K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe. 2008, November. "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, SC '08, 1–12. Austin, Texas, USA https://doi.org/10.1109/SC.2008.5214892.

Carothers, C. D., D. Bauer, and S. Pearce. 2000. "ROSS: A High-performance, Low Memory, Modular Time Warp System". In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS '00, 53–60. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc. https://doi.org/10.1109/PADS.2000.847144.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. "Efficient Optimistic Parallel Simulations Using Reverse Computation". *ACM Transactions on Modeling and Computer Simulation* 9(3):224–253 https://doi.org/10.1145/347823.347828.

Dickman, T., S. Gupta, and P. A. Wilsey. 2013. "Event Pool Structures for PDES on Many-Core Beowulf Clusters". In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, 103–114. New York, NY, USA: Association of Computing Machinery https://doi.org/10.1145/2486092.2486106.

Fujimoto, R. 1990. "Parallel Discrete Event Simulation". *Communications of the ACM* 33(10):30–53 https://doi.org/10.1145/84537.84545.

Fujimoto, R. M. 2000. *Parallel and Distribution Simulation Systems*. Wiley Series on Parallel and Distributed Computing. New York, NY, USA: John Wiley & Sons, Inc.

Gupta, S. 2018. *Pending Event Set Management in Parallel Discrete Event Simulation*. Ph. D. thesis, Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio. http://rave.ohiolink.edu/etdc/view?acc_num=ucin1535701778479768, accessed September 2019.

Jefferson, D. 1985. "Virtual Time". *ACM Transactions on Programming Languages and Systems* 7(3):405–425 https://doi.org/10.1145/3916.3988.

Lamport, L. 1978. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of ACM* 21(7):558–565 https://doi.org/10.1145/359545.359563.

Perumalla, K. S., and S. K. Seal. 2012. "Discrete event modeling and massively parallel execution of epidemic outbreak phenomena". *Simulation* 88(7):768–783 https://doi.org/10.1177/0037549711413001.

Shinde, G. S. 2024. "Efficient Synchronization and Input Queue Optimization in Parallel Discrete Event Simulation". Master's thesis, University of Cincinnati.

Tang, W. T., R. S. M. Goh, and I. L.-J. Thng. 2005. "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 https://doi.org/10.1145/1103323.1103324.

Watts, D. J., and S. H. Strogatz. 1998. "Collective dynamics of 'small-world' networks". *Nature* 393:440–442 https://doi.org/10.1038/30918.

Weber, D. 2016. "Time Warp Simulation on Multi-core Processors and Clusters". Master's thesis, University of Cincinnati, Cincinnati, OH.

AUTHOR BIOGRAPHIES

GAURAV SHINDE is currently part of Infection Prevention Team, R&D as Software Engineer at STERIS Inc. His current work is in area of embedded systems and optimizations along with *High Performance Computing*. His email address is gauravsanjayshinde@gmail.com or https://www.linkedin.com/in/gaurav-shinde-835620140/

SOUNAK GUPTA is currently a Principal Engineer and leads the low-latency telemetry development for cloud infrastructure at Oracle, Inc. He can be reached at sounak.besu@gmail.com or https://www.linkedin.com/in/sounakgupta/.

PHILIP A. WILSEY is a Professor in the Department of Electrical and Computer Engineering at the University of Cincinnati. His research interests are in *High Performance Computing* with applications to *Parallel Discrete-Event Simulation* and *Topological Data Analysis*. His email address is wilseypa@gmail.com and his research software is released from git repositories at: http://github.com/wilseypa.