Autonomous Orchestration of Distributed Discrete Event Simulations in the Presence of Resource Uncertainty

ZHIQUAN SUI, Colorado State University
MATTHEW MALENSEK, Colorado State University
NEIL HARVEY, University of Guelph
SHRIDEEP PALLICKARA, Colorado State University

Discrete event simulations model the behavior of complex, real-world systems. Simulating a wide range of events and conditions provides a more nuanced model, but also increases its computational footprint. To manage these processing requirements in a scalable manner, discrete event simulations can be distributed across multiple computing resources. Orchestrating the simulations in a distributed setting involves coping with *resource uncertainty*.

We consider three key aspects of resource uncertainty: resource failures, heterogeneity, and slowdowns. Each of these aspects is managed autonomously, which involves making accurate predictions of future execution times and latencies while also accounting for differences in hardware capabilities and dynamic resource consumption profiles. Further complicating matters, individual tasks within the simulation are stateful and stochastic, requiring inter-task communication and synchronization to produce accurate outcomes. We deal with these challenges through intelligent state collection and migration, active resource monitoring, and empirical evaluation of resource capabilities under changing conditions. To underscore the viability of our solution, we provide benchmarks using a production discrete event simulation that can simultaneously sustain failures, manage resource heterogeneity, and handle slowdowns while being orchestrated by our framework.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Fault tolerance; I.6.8 [Simulation and Modeling]: Discrete Event

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Fault tolerance, distributed discrete event simulation, checkpointing, neural networks, prediction

1. INTRODUCTION

In situations where direct experimentation is difficult, expensive, or simply infeasible, discrete event simulations (DES) provide a highly *expressive* solution for modeling real-world systems. This expressivity is derived from representing the interactions between all components in the system as *events*. Accounting for these interactions is often compute-intensive, which makes a divide-and-conquer approach over several computing resources an ideal means for quickly gaining insights from the simulation. Dividing the workload is beneficial in multiple ways: outputs can be generated faster, more parameters can be explored, and additional iterations of the simulation can be run to verify output quality. We have explored the challenges involved with performing load balancing to optimize performance in the context of a discrete event simulation in a previous study [Sui et al. 2013]. However, these performance gains come at the expense of additional complexity through increased communication and

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018) and the US National Science Foundation's Computer Systems Research Program (CNS-1253908). Author's addresses: Z. Sui, M. Malensek, and S. Pallickara: Department of Computer Science, Colorado State University, Fort Collins, CO, USA; N. Harvey: School of Computer Science, University of Guelph, Guelph, ON, Canada.

synchronization required between distributed components. As more components are added to a system and executed across a diverse set of computing resources, the likelihood of a failure also becomes much higher.

The primary theme of this paper is *resource uncertainty,* of which we consider three key aspects: **fault tolerance, resource slowdowns,** and **heterogeneity**. Ultimately, each of these items can have severe and unexpected performance consequences in a distributed setting and must be accounted for to ensure resources are used efficiently. Our prior research that focused on autonomous fault tolerance functionality [Malensek et al. 2013] has been extended in this paper to target the two remaining aspects of resource uncertainty. ***Resource slowdowns*** may occur due to an increase in the number of processes executing concurrently at the resource, load spikes, or runaway processes resulting from a coding error. If not circumvented, the consequences of such slowdowns can lead to extended execution times. Accounting for ***resource heterogeneity*** is critical especially when apportioning tasks to resources with the objective of faster completion times. In this work, we consider the performance-centric aspects of heterogeneity to achieve greater system throughput. Our algorithms identify resource slowdowns and circumvent them by launching speculative tasks (duplicate tasks that essentially "race" the original task to completion). These speculative tasks are launched on the best available hardware configurations to ensure faster completion times.

In the modern computing landscape, failures are an issue that must be expected and dealt with rather than simply avoided or ignored [Patterson et al. 2002]. The longer a software process runs, the more likely it is to experience a hardware or software failure. This fact is only exacerbated by the trend towards distributed, multi-core architectures that take advantage of the vast processing power found in today's commodity hardware. Each additional unit of processing involved in a task increases the overall probability of a failure occurring.

Our approach to mitigating resource failures involves designing an autonomous fault tolerance agent to oversee the execution of distributed discrete event simulations. Unlike conventional distributed fault tolerance approaches, our system must cope with constantly changing, stateful computations and ensure global consistency throughout the system in the event of a failure. This requires system-level optimizations along with reliable prediction mechanisms that enable a dynamic, proactive approach to providing fault tolerant execution for our subject simulation. We rely on an adaptive strategy that determines when and how checkpoints should be requested in order to reduce the amount of duplicate work and overhead incurred from state collection. This is made possible by forecasting state changes and having the system plan accordingly.

Slowdown of a resource results in tasks executing on that resource executing significantly slower than other tasks executing concurrently on other resources. Resource slowdowns must be mitigated in a timely fashion because of their cascading effects. Discrete event simulations include synchronization barriers where different components of the simulation synchronize their state. Between successive synchronization barriers, the execution time is only as fast as the slowest task. This means that a single slow task greatly influences the overall execution time.

To mitigate resource slowdowns we rely on detection and circumvention. We predict resource slowdowns using artificial neural networks based on several factors, including the number of context switches, number of collocated processes, memory

consumption, and paging. We use speculative tasks to alleviate slowdowns by launching slow tasks on faster machines.

Resource heterogeneity in distributed settings occurs naturally as a result of how machines comprising a cluster are added, upgraded, and retired. Coping with resource heterogeneity requires us to rank resources and continually monitor them to detect slowdowns. A critical aspect in dealing with resource heterogeneity is how tasks are apportioned. Large, compute-intensive tasks must be launched on faster machines. Such apportioning is needed to minimize imbalances between the tasks that comprise a simulation, resulting in faster completion times.

Paper Contributions

In this paper we describe a holistic approach for timely, distributed orchestration of discrete event simulations in the presence of resource uncertainty. Our specific contributions include:
(1) Use of a learning-based, adaptive checkpointing strategy, (2) ability to handle multiple concurrent failures, both persistent and transient, (3) alleviation of resource slowdowns in a timely fashion: we predict impending resource slowdown using artificial neural networks and launch speculative tasks to circumvent them, (4) minimization of duplicate processing by launching speculative tasks only for those that are likely to impact completion times, and (5) harnessing resource heterogeneity by monitoring and ranking resources, and subsequently using this in our task apportioning scheme to minimize imbalances between subtasks and faster completion times.

2. SYSTEM ARCHITECTURE

Our system is designed to separate concerns between two components: the cloud runtime, which handles orchestration of the simulation, and the simulation itself. This separation allows different discrete event simulations to be run within the system, provided that they conform to our wire format specification. The wire format describes how state information should be shared among components. In this study, we used the Granules Cloud Runtime [Pallickara et al. 2009] for coordinating distributed activities within the system, and the North American Animal Disease Spread Model (NAADSM) [Harvey et al. 2007] as our subject discrete event simulation.

2.1 Granules

Granules [Pallickara et al. 2009] is a distributed stream-processing framework that allows computations to be expressed using the MapReduce paradigm or as directed, cyclic graphs. The framework handles deploying, scheduling, and orchestrating computations on clusters of machines or in the cloud. Computations can be scheduled to run when data is available or at regular intervals, with a configurable number of execution iterations. Since computations can execute multiple times, it is possible to build state over the course of execution. Granules has been employed in several areas of study, including bioinformatics, brain-computer interfaces [Ericson et al. 2010], clustering [Ericson and Pallickara 2012], and scientific data storage [Malensek et al. 2012].

While Granules is implemented in Java and natively supports Java-based computations, the framework also provides bridging functionality that allows computations to be written in C, C++, Python, and R. NAADSM is written in C and

uses this functionality to communicate directly with the Granules runtime. Granules is an open source effort, available at http://granules.cs.colostate.edu.

2.2 NAADSM

NAADSM [Harvey et al. 2007] is an epidemiological model of disease outbreaks in livestock populations developed jointly by the US Department of Agriculture, the Canadian Food Inspection Agency, Colorado State University, the University of Guelph, and the Ontario Ministry of Agriculture, Food, and Rural Affairs. It has been applied to studies of several diseases including foot-and-mouth disease [Pendell et al. 2007], avian influenza [Green et al. 2010], and pseudorabies [Portacci et al. 2009].

NAADSM is a Monte Carlo model: a simulation is run many times with each run representing one possible way that events could occur in a disease outbreak, which contributes to an overall picture of the probability distributions of the output variables. This means that simulations are generally run many times, and due to their processor-intensive nature can benefit greatly from a parallel, divide-and-conquer approach spread across multiple computing resources. Like many other discrete event simulations, NAADSM represents a complete iteration of its event processing loop as a basic unit of time; in our case, this is the *simulation day,* which maps to real-world passage of time to model how long a disease outbreak might last. We refer to this construct as a *simulation interval* in the text, as it could be applied to minutes, hours, or even years depending on the particular simulation being used.

In this study we used a NAADSM *scenario* that simulated an outbreak of foot-and-mouth disease (FMD). The population was based on Kansas, USA data, but up-sampled from 46,000 to 660,000 farms, which is the approximate number of FMD-susceptible farms in the 12 Midwest US states. The scenario's scale and parameters were selected specifically to produce computationally intensive simulations; on a single machine in our test cluster, the scenario takes about 18 hours to execute.

2.3 Test Environment

The benchmarks described in this study were carried out on a 78-node cluster consisting of 48 HP DL160 servers (Xeon E5620, 12 GB RAM) and 30 HP DL320e servers (Xeon E3-1220, 8 GB RAM). Each machine was equipped with a gigabit network interface, and single-core benchmarks were performed on the DL160 models. The Granules framework ran within OpenJDK 1.7.

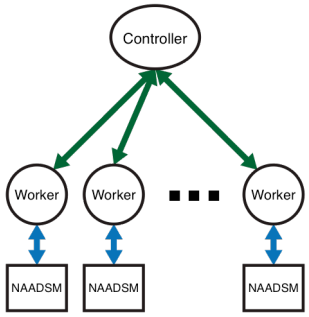

Figure 1. The network layout for our distributed discrete event simulation framework.

2.4 Parallelization Framework

A distributed execution of NAADSM across a cluster of computing resources involves two primary components: a Controller, which coordinates the global state of the simulation, and multiple worker instances, which are deployed to each resource in the cluster and are responsible for managing individual instances of NAADSM. Figure 1 illustrates this configuration. These two components are the backbone of our discrete event simulation parallelization framework.

When running within Granules, a NAADSM simulation is divided by geography with each worker managing a subset of farms in the scenario. Events that may have effects outside of a worker's territory are packaged and transmitted to the Controller

in a once-per-simulation-day step that acts as a synchronization barrier across the workers. After receiving state updates from each worker instance, the Controller broadcasts an aggregated bundle of updates out to the workers to mark the start of a new simulation interval. Barrier synchronization is one of several possible methods for parallelizing an event loop based discrete event simulation, and has the advantage of requiring minimal modification to the underlying simulation architecture. In other words, our framework strives to treat the simulation as a black box. Since we do not break the simulation into several discrete logical processes (LPs), each worker runs the same process with different input and state parameters. This allows us to divide a simulation up without modeling interdependencies.

To balance load across the available resources in a cluster, our system employs a dynamic split and merge strategy to divide or consolidate workers depending on changing load characteristics. Due to the simulation interval execution barrier, it is critical to place load as uniformly as possible across all the workers in the system; the simulation can proceed only after all workers have completed their tasks, so a run of a simulation interval is only as fast as the slowest worker.

Our parallelization framework is best suited for simulations that have a spatial component that can be divided into individual sub-regions managed by a worker instance. This includes atmospheric science [Cotton et al. 2001], modeling object interactions in space [Jefferson and Leek 2010], and cosmology [Springel 2005].

These architectural decisions make the distributed orchestration layer highly adaptable to new types of simulations, but also make each component instance a single point of failure; the loss of either the Controller or even a single worker prevents further progress of the simulation. To deal with these failure scenarios we have introduced another component, called the Speculator, which handles speculative execution, failure detection, and recovery operations from within the cloud runtime.

Unlike the speculative execution performed in implementations of MapReduce [Dean and Ghemawat 2008], such as Hadoop [Bialecki et al. 2005], our system requires transactional semantics to ensure the consistency of global state across the entire cluster during failures. Additionally, the stochastic nature of our problem leads to the development of execution hot spots that move across different processing elements during the simulation, precluding the sole use of execution time as a failure or slowness metric.

3. COPING WITH RESOURCE FAILURES

There are several challenges in ensuring autonomous execution of discrete event simulations in the presence of resource failures. These include:

- *Computations composing the simulation are stateful*: Though the simulation is stochastic, outcomes depend on the state built up at each task in the simulation.
- *Simulation stalls during failures*: State coordination is performed at synchronization points (the end of a simulation interval). Here, the failure of a single machine can stall the entire simulation.

3.1 Research Questions

Creating a fault-tolerant, distributed implementation of a discrete event simulation that requires stateful, interdependent tasks led us to pose a number of research questions that we have addressed in this paper:

1. Can we incorporate support for failure resilience while minimizing overheads? Since failures are infrequent, our goal should be to achieve fault tolerance without introducing unacceptable overheads into the system.
2. Can these overheads be minimized while also retaining the ability to cope with multiple, concurrent failures, which can either be permanent or transient?
3. Can failures be detected efficiently? Simply executing duplicate tasks in parallel is not an adequate scheme for dealing with failures in our system, so active detection of failures is required.
4. Will simply collecting state information from individual tasks be efficient means to provide fault tolerance? How can state collection be optimized?
5. In the event of a failure, can we minimize the amount of duplicate work?

3.2 The Speculator

In our system, the Speculator is responsible for all activities related to fault tolerance and speculative execution. This includes detecting failures, launching new workers, rolling the simulation back to a consistent state, and managing resources. The Speculator can also be used to suspend a simulation, save its state to disk, and then resume execution — even on completely different hardware.

Along with these fault tolerance features, the Speculator also plays a role as an autonomous manager of system events, deciding the frequency of system state collection and allocation of resources that can be used by the Controller. These decisions allow the Speculator to provide its services without imposing a significant performance penalty on the execution of the simulation.

One key aspect of our system architecture is that communication only occurs once per simulation time unit, which is a simulation day in the case of NAADSM. This means that querying a worker and receiving a result is a two-day process. Because of this constraint, the Speculator must be highly proactive; simply reacting to events as they take place is not sufficient due to the ever-changing state of the simulation.

As a simulation progresses, state updates are accumulated during each simulated day. In the case of a failure at any resource, the simulation cannot progress any further because all workers must report state updates that may have effects outside their own territory before continuing. The challenges that arise in this situation are twofold: detection of failures, and failure recovery.

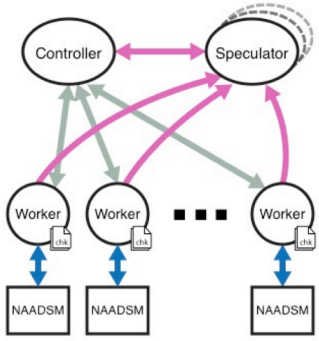

Figure 2. Communication between components and the Speculator, with heartbeats and state transfers highlighted.

3.2.1 Failure Detection

To facilitate failure detection, the components in our system transmit small messages, called *heartbeats*, to the Speculator at regular intervals. Heartbeats contain system statistics, such as load information, CPU utilization, available memory, and disk activity, which are used when making fault tolerance decisions. The interactions between the Speculator and the rest of the system are shown in Figure 2.

When a worker or Controller instance has not sent a heartbeat message after a configurable failure timeout threshold, the Speculator assumes it has failed. This assumption is confirmed by instructing one of the other machines in the system to

also attempt to contact the resource. In our tests, a heartbeat interval of 5 seconds resulted in a failure being detected in about 7.82 seconds (averaged over 1,000 samples). This frequency is determined by the average completion time for a simulation interval.

To handle situations where a Speculator fails, multiple instances are started transparently and operate in parallel as *Shadow Speculators* that also receive heartbeat and checkpoint messages. Shadow Speculators incur additional network overhead due to message duplication, so launching two instances generally provides a reasonable balance between failure resilience and performance. The Speculator replication factor (such as **3**) is automatically maintained by dynamically starting additional instances as necessary.

3.2.2 Failure Recovery

Workers in our system maintain both local state and global state. Global state information is exchanged during each new simulation interval, but local state is only used at the individual worker level. With no fault tolerance considerations, this leads to an unrecoverable simulation in the event of a worker failure. To cope with the possibility of a worker failure, local state information is sent to the Speculator as a *checkpoint*. Checkpoints contain enough local state information for the Speculator to re-launch a worker process. Unfortunately, simply re-launching a worker is not enough to resume a simulation; all workers must be executing the same simulation interval, so the entire simulation must be rolled back to a consistent state. This process may require starting or stopping worker instances.

Since the Speculator monitors the resources' system status information and the simulation state as well, it can reason about what information it will need to provide fault tolerance without impacting the overall performance of the system. For example, given a particular disease, the simulation may progress rapidly and then slow down as the disease spreads and becomes more computationally intensive. Figure 3 illustrates this situation, showing execution times in a 64-worker simulation run of our Midwest scenario. In the early stages of a simulation, the Speculator does not need to request checkpoints frequently because a restart of the entire simulation would have a minimal impact on overall execution time. This property makes setting a hard "checkpoint interval" inefficient, and led us to develop a fault tolerance component designed around making intelligent, autonomous decisions about ***how*** and ***when*** checkpoints should be collected.

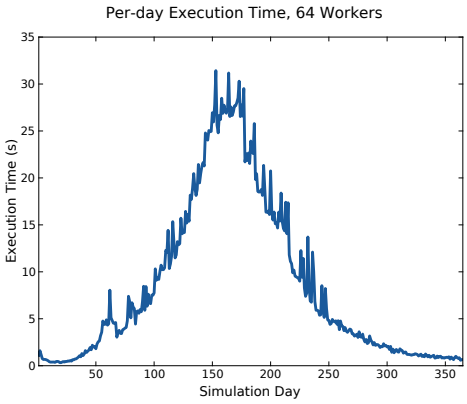

Figure 3. Per-day execution time of a 64-worker Simulation of foot-and-mouth disease in Midwest USA.

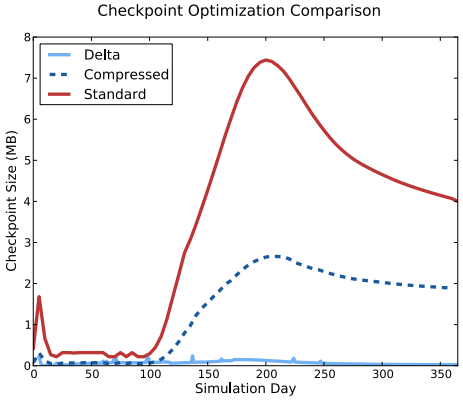

Figure 4. Per-worker checkpoint sizes for each checkpointing strategy.

3.3 Checkpoint Optimizations

Checkpoints are used for two primary purposes in our system: load balancing and fault tolerance. As one might expect, the amount of state information in a simulation tends to grow over the course of execution, with the most growth seen during highly complex events. We have observed that the most CPU-intensive portions of a simulation are accompanied by a growth in checkpoint sizes.

Given a 64-worker run of the simulation, individual checkpoints of about 8 MB will result in 512 MB of data being sent to the Speculator at a time, easily saturating its network interface. Decreasing the size of these checkpoints allows the system to collect state information more often, reducing the amount of simulation progress lost in the event of a failure. We implemented two approaches to reduce the overall size of messages being sent in the system: *transparent compression*, and *delta checkpoints*.

3.3.1 Transparent Compression

With transparent compression, messages sent through Granules are automatically compressed using the DEFLATE algorithm [Deutsch 1996], which provides a balance between speed and output file sizes. Compressing small checkpoints proved to have only a marginal benefit (10% reduction in size), but larger 4-7 MB checkpoints had a compression ratio of about 0.35, and took about 65 ms per megabyte to create.

3.3.2 Delta Checkpoints

The current state of a discrete event simulation represents how a sequence of events has unfolded thus far. The progression of these events causes state changes in an iterative fashion, meaning each checkpoint in our system has some common attributes with previous checkpoints. To exploit this property, we created delta checkpoints, which contain the binary differences from the last checkpoint generated.

Several algorithms have been developed for generating binary patch files, such as VCDIFF [Korn and Vo 2002], XDelta [MacDonald 2008], and bsdiff [Percival 2006], which are frequently used in the distribution of software updates. For this work, we used our native Java implementation of bsdiff because of the small patch files it produces, even when there is a large time gap between checkpoints (multiple simulation intervals, in our case). To further speed up the delta checkpoint creation process, we replaced the bzip2 compression used in bsdiff with the same DEFLATE algorithm used in transparent compression. While bzip2 offers better compression ratios, it also tends to consume more processing time. Table 1 summarizes the delta checkpoint sizes and their creation times.

Ultimately, there are tradeoffs associated with each of our checkpointing methods. Small checkpoints generally do not need further compression or patching. Larger checkpoints that are created during fast-executing parts of the simulation benefit most from compression; the compressed files save network bandwidth and can be generated quickly. During the longest-running portions of the simulation, delta checkpoints enable the system to collect state information frequently to mitigate the large amount of lost processing time a failure would cause. Figure 4 shows the size differences between uncompressed, compressed, and delta checkpoints.

Table 1. Delta checkpoint generation averaged over 1000 runs.

Checkpoint Size (MB)	Delta Size (KB)		Creation Time (ms)	
	Mean	Standard Deviation	Mean	Standard Deviation
1	38.66	3.53	517.26	79.07
4	116.01	4.16	1037.43	36.57
7	126.40	9.72	1579.68	56.48

3.4 Fault Tolerance Policy

The Speculator has several options at its disposal for providing fault tolerance, each with their own tradeoff space. *How* and *when* the Speculator requests checkpoints ultimately determines the performance impact of our approach and its scalability as more workers are added. While thresholds can be set to guide the Speculator's choices, hard rules tend to be brittle and ineffective when faced with different simulation and disease types. To provide a flexible model for determining the appropriate checkpointing policy, we predict the per-interval execution time of the simulation, as well as the cost of generating a checkpoint. These future costs can then be evaluated against the likelihood of failures to produce a fault tolerance strategy.

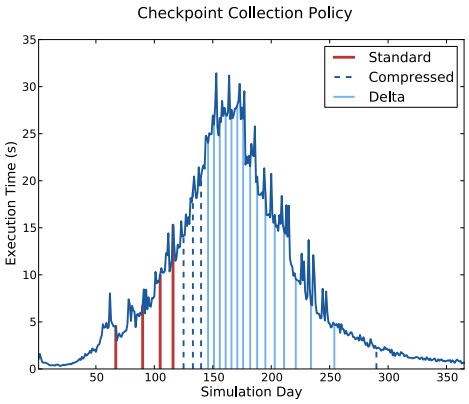

Figure 5. Prediction-based checkpoint policy changes over time. Days that checkpoints were requested on are indicated by a vertical line.

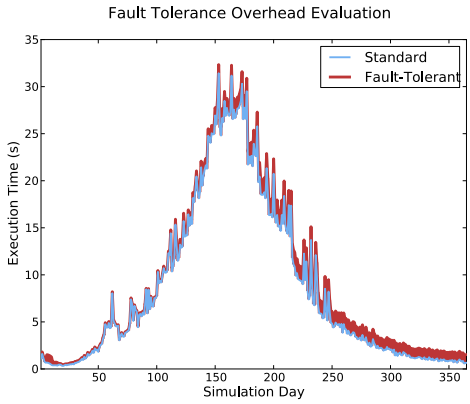

Figure 6. Shaded areas above the standard execution times represent overhead from checkpointing.

3.4.1 Speculative Prediction: Artificial Neural Networks

Since execution times in our simulation tend to exhibit non-linear patterns, we used an Artificial Neural Network (ANN) from the Encog Machine Learning Framework [Heaton Research 2013] to enable the Speculator to predict execution times and checkpoint costs. ANNs consist of multiple interconnected *neurons* that receive inputs and produce outputs based on an *activation function*. These functions are influenced by weights that are derived by *training* the network with representative data and adjusting the weights accordingly. ANNs have an input layer and output layer; in our case, the inputs include execution times, checkpoint sizes, resource usage statistics, and simulation state variables, while the outputs are the predicted execution time and checkpoint sizes. One or more *hidden layers* exist between the input and output layers, and can be configured with a number of hidden units (neurons) to enable a variety of functions and patterns to be learned.

Collecting a complete snapshot of the system state requires at least two simulation intervals: one to request the checkpoints, and one to receive them. Due to this constraint, we trained the neural network to predict three simulation intervals into the future. This gives the Speculator time to choose an appropriate checkpointing strategy for upcoming events and inform workers of when the checkpoints should be generated. Both the data contained in heartbeat messages and simulation state variables were fed into the neural network. Figure 5 illustrates how and when the Speculator requested checkpoints in an iteration of our Midwest

scenario. The policy ensured that no more than two minutes of execution time could be lost at any point due to a failure.

To illustrate the advantage of using our adaptive checkpointing strategy, we compared its fault tolerance overhead with three simulation runs that had a static checkpoint interval of **3** and a fixed checkpoint type. Table 2 shows the stark difference in overhead between a static and adaptive strategy. The table also demonstrates that simply using a delta strategy for the smallest possible checkpoint size does not result in the lowest overhead of the fixed strategies. The slight increase in overhead compared to compression is due to the additional processing time required to compute delta checkpoints.

Table 2. Comparison of adaptive and fixed checkpointing strategies.

Strategy	**Overhead**
Fixed, Standard	7.32%
Fixed, Compressed	5.8%
Fixed, Delta	5.98%
Adaptive	***1.1%***

Table 3. Failure recovery times at different stages of a scenario's execution.

Failure at Interval	**Recovery Time (s)**	
	Mean	**Standard Deviation**
50	5.5	1.3
200	10.4	2.7
300	7.9	2.1

3.4.2 Fault Tolerance and Failure Recovery Overheads

In order to evaluate the worst-case performance overhead incurred from our fault tolerance system, we ran two identical iterations of our Midwest scenario: one with no fault tolerance functionality enabled, and one with checkpoint requests occurring every simulation interval. Figure 6 shows the per-interval cost of collecting checkpoints in our system; the overhead accounted for a five-minute increase in execution time, but the majority was during non-critical execution at the end of the simulation. These costs can be avoided with accurate forecasting, as our benchmarks have shown.

Table 3 contains recovery overhead information for induced failures at simulation intervals (days) 50, 200, and 300. These overheads are incurred after the failure has been detected, and the majority of the cost is network I/O from transferring state.

4. CIRCUMVENTING RESOURCE SLOWDOWNS & MANAGING HETEROGENEITY

When simulations execute over a collection of resources, uncertainty stems from: resource heterogeneity, how tasks are apportioned among heterogeneous resources, load spikes at the resources, and load variations due to the information exchange at synchronization barriers within a simulation.

It is not unusual for a resource to have spikes in the number of processes that are executing concurrently, leading to increased contention and thus strains on CPU, memory and disk utilization. We need to be able to detect such resource slowdowns and circumvent them via apportioning of tasks or by launching speculative tasks that duplicate the processing being performed on the slow resource. Our experiments in this section involve a 64-worker deployment, where workers are dynamically migrated to any machine in our 78-node cluster (which may result in several workers residing on a single machine).

Since multiple workers execute each simulation interval in parallel, *imbalances* may exist. Tasks that complete earlier must wait until the slowest task completes; only then is it possible to synchronize state among the tasks and proceed to the next simulation interval. In general, speed and throughput drop as imbalances occur.

4.1 Challenges

Our research objective is faster, distributed execution of simulations in the presence of such uncertainty. There are several challenges involved in accomplishing this:

- *The number and frequency of synchronization barriers is high*: Distributed simulations often have multiple synchronization barriers during the course of execution. These synchronization barriers are used to exchange information about state-changes within particular subtasks that have an impact on the entire simulation. In our example case of epidemiological modeling, subtasks synchronize their simulation state at the end of each simulation interval.
- *Apportioning tasks among heterogeneous resources*: Since the simulation is stochastic, the time spent by each task on a particular simulation interval varies. The execution times cannot be calculated in a deterministic fashion either before or during the simulation run. Apportioning of tasks must account for two objectives: (1) minimization of imbalances and (2) avoidance of resource slowdowns.
- *Detecting and circumventing resource slowdowns***:** There is no way to determine what percentage of a task has been completed: once a task begins work on a particular simulation interval, the only information available is whether the task is still executing, has completed, or has failed. Slowdown detection must be done as the task is executing and must be inferred based on factors such as CPU utilization, memory availability and utilization, disk I/O, and network I/O.
- *Mitigating imbalances in workloads*: Imbalances may exist between synchronization barriers. These imbalances may either be due to the scope of a particular task or the machine that it is executing on. If an attempt is not made to mitigate these imbalances, they will persist for the next synchronization barrier and could get worse, leading to longer completion times.
- *Interference uncertainty*: The interference (due to other concurrent processes) at a machine is unpredictable. This interference could be periodic or bursty, and can occur at any machine at any time. The greater the interference, the greater the CPU and memory contention, leading to correspondingly longer execution times.

4.2 Research Questions

Research questions that we explore in the context of orchestrating simulations in the presence of resource heterogeneity and slowdowns include the following:

- There is a mix of resources in the system. Can we profile these resources and rank them accordingly?
- How can we apportion tasks so that we exploit this heterogeneity? Can we distribute tasks across resources such that the total execution time for a simulation interval is minimized?
- How can we cope with resource slowdowns? To accomplish this we must be able to identify slowdowns and initiate counter measures to mitigate them.
- How can we ensure fast completion times in such settings?

4.3 Methodology: Circumventing Slowdowns and Coping with Heterogeneity

Our approach targets several aspects of coping with resource uncertainty. These include: profiling resources, apportioning tasks while taking into account both the complexity of the task and resource profile, identification of resources that are slowing down, and launching speculative tasks on faster resources to circumvent

stragglers (tasks still working on a simulation iteration while most of the others have finished).

We gather resource capabilities such as the number of cores and available, physical memory, and along with a software benchmark use these profiles to normalize and rank resources. We use this ranking of resources to launch compute-intensive tasks on best available resources, which are likely to provide the shortest completion times for the task. We also launch straggler tasks based on this resource ranking.

We apportion tasks over the available resources based on the expected completion times for tasks between synchronization barriers. The objective of this apportioning is to minimize imbalances between tasks, *i.e.*, to ensure that tasks complete at more or less the same time between synchronization barriers. Tasks that are likely to be compute-intensive and long running must be executed on faster machines.

Since the simulation is stochastic, we cannot deterministically calculate the task completion times; rather, we need to predict them. We predict execution times at a fine-grained level: for each task and for each simulation interval. We also predict completion times for tasks between synchronization barriers. For the simulation that we consider, this prediction is based on disease biology (incubation period, infectious period, etc.), geographical scope of the region being managed by the particular task, and the disease prevalence within the region at the particular synchronization barrier. We use artificial neural networks to make this prediction.

Next, we identify tasks that are likely to impact execution time. These could be tasks that have a disproportionate share of the processing workload (even though they may execute on faster machines) or may be executing on slower machines. This allows us to identify tasks that are likely to take longer to execute and introduce imbalances where other tasks must wait at the synchronization barrier. We use the synchronization barriers to gather this information and also to apportion workloads to mitigate imbalances.

Another issue that needs to be accounted for is *stragglers*, which are tasks with longer than average completion times. There are two aspects related to dealing with stragglers. First, we need to identify tasks that are likely to be stragglers. Second, we must circumvent stragglers by launching speculative (backup) tasks on faster resources.

We use *speculative tasks* as a mechanism to mitigate resource slowdowns. Doing so will require us to solve three problems: the *which*, *where* and *when* problems. Deciding *which* tasks to launch will involve detecting which workers are running more slowly than expected and which are likely to slow down soon. Deciding *where* to launch the tasks will involve resource state prediction and considerations around data locality and network topology. Deciding *when* to launch will involve solving a tradeoff: we want to respond swiftly to changing conditions, but avoid excessive overhead from too-frequent launches.

To identify tasks that are likely to be stragglers, we predict when resources will slow down. We use an ANN to predict slowdowns based on tracking CPU utilization, the number of processes, and memory utilization including paging. Since execution times also increase because of an increase in CPU-bound processes that result in increased context switches, we track them as well.

We then launch speculative tasks for stragglers on better machines. Our approach identifies: (1) a set of machines that have spare capacity to take on speculative tasks, and (2) the stragglers that are most likely to benefit the simulation (*i.e.*, faster

executions of these tasks would decrease imbalances and result in shorter overall execution time). We also consider situations without interference and quantify the overhead added by the speculative launch mechanism.

4.4 Profiling resources

In our public cluster, resources are heterogeneous and we wish to launch tasks on the best available resources. We rank resources by CPU speed: our simulation is a CPU-intensive task, so memory and disk I/O are not expected to be the dominating factors. We measured CPU speed with a benchmark called the RAW benchmark suite. Because we are using a public cluster, we attempted to avoid having other users' processes interfere with the speed measurement by 1) selecting a benchmark from the suite that has a short running time (we chose the FFT benchmark), 2) running the benchmark at several different times of day, and 3) choosing the fastest recorded time. We hypothesize that the fastest test is the one with the least interference.

Beyond CPU speed, we also record CPU utilization, memory and network use, and disk I/O. Each instance of our test simulation uses a single CPU core and a predictable amount of memory, which provides a baseline estimate of the cost of launching another worker process. The criteria we developed to identify resources that can run a worker instance without interfering with other users' processes are given in the following subsections.

4.4.1 CPU

We account for both user space processing (UserTime) and kernel processing (SystemTime). We also measure the number of active processes (load average). The constraints we developed are (percentages taken across all cores):

- SystemTime <= 35%, UserTime + SystemTime <= 70%
- Load Average / Total number of cores <= 3

4.4.2 Memory

We consider both existing tasks that may be running in resource-constrained situations as well as the conditions for scheduling new tasks. Acceptable memory conditions are:

- Memory Consumed <= 70% (existing tasks)
- Memory Consumed <= 50% (new tasks)

Additionally, we inspect the percentage of I/O requests related to swap operations (paging). If swapping accounts for even a small portion of I/O, then the resource lacks memory. Therefore, we only consider resources that are not actively swapping.

4.4.3 Network and Disk I/O

For disk-based I/O, we inspect average disk utilization over a configurable window and favor machines at or below **30%** of their capacity. We also evaluate the average waiting time of all disk requests (*await*). We ensure that the network latency between components is low with a configurable threshold. In our cluster, we considered machines with a ping latency above **5ms** to be experiencing slow network conditions.

4.5 Harnessing Resource Profiles

Our algorithm is built around a sorted list in which each node represents a machine, and the list is sorted by the *current speed* of the machine. The current speed is the

product of the "normalized speed" (as measured by the benchmark program) and a *slowdown factor*. The slowdown factor represents the ratio between performance with and without resource contention by monitoring long-term trends. The long-term trend is the measured historical pattern of usage at certain times of the day.
Each node of the list contains the items below:

1. Number of available and occupied resources
2. Hardware information (CPU, Memory, Disk I/O, Network I/O)
3. Normalized speed α_i and slowdown factor S_i.

The Controller and Speculator use this list to find the fastest machine. When launching speculative tasks, the system will compare the predicted execution time of the task on the current machine and the fastest spare machine in the list. Since the list is sorted during each update, determining the appropriate machine is fast.

4.5.1 List updating mechanism

The list is updated frequently, with the timing of the updates based on two *heartbeats*. The more frequent heartbeat occurs every 10 seconds, and it triggers a check of the CPU use, memory use, network use, and disk I/O. If the memory use, network use, or disk I/O is over threshold as defined in the previous section, then the slowdown factor is set to 0. A slowdown factor of zero will make the computed *current speed* zero, thus placing the machine at the bottom of the sorted list and preventing new speculative tasks on the machine. However, there is one exception to this rule when a resource has much greater capabilities than the baseline (as detected by our benchmark). In these situations, the slowdown factor will not be zeroed if the resource in question has unused capacity equal or greater than the baseline hardware configuration. After these aspects have been considered, the measures of CPU use are combined into a vector and fed to an ANN to predict the slowdown factor.

The less frequent heartbeat occurs every 5 minutes, and it triggers a check of the historical trends data. We measured the usage of all machines in our cluster for 24 hours to discover patterns. For example, some machines are used for classes during the daytime, but are idling at night. There are also programs launched at specific times of the day by administrators and by other research groups. If the historical usage data shows that the machine will soon be occupied, then the slowdown factor is set to zero.

4.5.2 Launching speculative tasks

Before launching a speculative task, we consider whether the operation is likely to provide a performance gain. Launching a speculative task will impose overheads, specifically, transferring a checkpoint over the network. We assume the network transfer latency for checkpoint has a linear relationship with the checkpoint size.

The predicted time for a task to run on the current machine i is represented as T. Given the normalized speeds and the slowdown factors found for machines i and j, and the transfer latency T_{trans}, then the predicted time to transfer a checkpoint to machine j and execute the task on machine j is $\frac{\alpha_j S_j}{\alpha_i S_i} T + T_{trans}$. If $\frac{\alpha_j S_j}{\alpha_i S_i} T + T_{trans} < T$, we launch the speculative task and we predict the performance will be better.

There are two ways the system decides to launch speculative tasks. Between simulation intervals, the system will decide whether or not to launch speculative tasks. At this time, speculative tasks can start at the same time as the normal tasks, avoiding additional latency costs. The second way to launch a speculative task is

based on the heartbeat that checks machines' current speed. If a machine is operating slower than our system would expect, we predict whether performance would be improved by launching speculative tasks for the all of the tasks currently on the slow machine onto the fastest machine. This helps avoid toggling, where loads constantly shift between two machines.

Our system already contains a fault tolerance process in which checkpoints are requested when necessary. However, the slowdown detection mechanism requires a checkpoint for each simulation interval to enable fast and dynamic migration of processes. Therefore, workers also store a *local checkpoint* on their disk after each synchronization point. Local checkpoints only incur network latency in the event of a process migration, and are handled by a separate thread to avoid blocking the simulation process. This also means that when the Speculator requests a checkpoint, the latest local checkpoint will be transmitted without needing to be generated on-demand. This feature allows us to dynamically manage the tradeoff space between I/O overhead and checkpoint availability.

When speculative workers are launched, they get added to the worker list alongside the other workers that are responsible for the same task. Whichever worker finishes the task first will be kept active for subsequent simulation intervals; the other workers that were managing the same task are "cleaned up" and returned to the worker pool. When requested, workers can directly transmit their state information to another worker instance to create a speculative task. This avoids the need for centralized communication through the Controller or Speculator.

4.5.3 Execution Time Prediction

We use an additional Artificial Neural Network to predict the slowdown factor. The inputs to the ANN are the number of processes on a machine, number of users, load averages, and CPU time (system, user, and idle). The output from the ANN is the slowdown factor discussed at the beginning of this section. The ANN was configured with one hidden layer with 15 hidden units.

Data to train the ANN was gathered with the RAW benchmark and our own monitoring tools. CPU speed was first measured on various machines without load, and then the test was repeated while random loads were placed on the machines. We used resilient propagation (RPROP) to train the network.

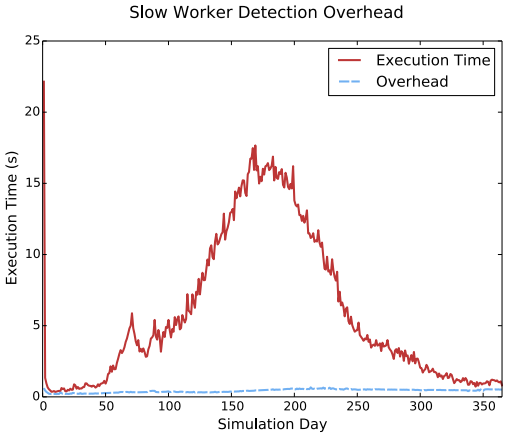

Figure 7 Execution time vs. overhead

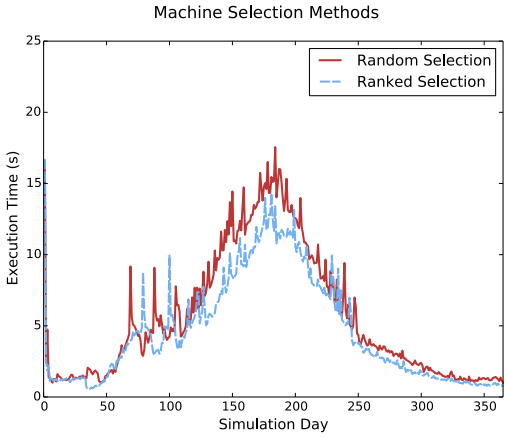

Figure 8 Ranked selection comparison

4.6 Performance evaluation of managing heterogeneity and circumventing slowdowns

As a first step, we measured the overhead of our slow worker detection mechanism. In comparison with the original Granules-NAADSM framework, the new framework adds a checkpoint operation every simulation interval. This new step could have a negative impact on execution time.

In this experiment, each worker requested a complete checkpoint message from the NAADSM simulator and stored it on the local disk. There is no network communication or data compression used in this process. We measured the time gap between the checkpoint request and feedback from the simulation. The overhead versus execution time is shown in Figure 7. We see that the overhead time is almost constant, around 500ms for all simulation intervals. In comparison with the execution time of more than 5 seconds on an average simulation interval, this overhead is small.

Our test environment was a cluster of heterogeneous machines, with considerable variation in CPU speed, total memory, and network speed. During the experiments, there might or might not be interference from other users. In our experimental design, we show results from both situations. Even if there is no interference from other users, our new framework can still provide improved execution times. In previous versions of our distributed orchestration system, worker processes were launched in a random pattern. With the slowdown detection mechanism, machines are ranked by current speed, and so worker processes will be launched onto the fastest available machines. The comparison of performance with random machine selection vs. selection of the fastest machine is shown in Figure 8. The total execution time with random machine selection is 2034.5 seconds, and with selection of the fastest machine is 1684.7 seconds, an improvement of 17.2%. The peaks around simulation day 100 occur because there is a *merge threshold* of 5 seconds. The system does not do merge operations when the execution time is less than 5 seconds to avoid a situation where split and merge overheads dominate the execution time. As shown here, slowdown detection works well even when there is no interference from other users of the cluster.

In the next experiment, we added interference into the cluster by launching resource-intensive programs on specific machines before starting the simulation. By avoiding busy machines with the machine selection functionality, the split-merge strategy performs well even without the speculative launch mechanism. This is

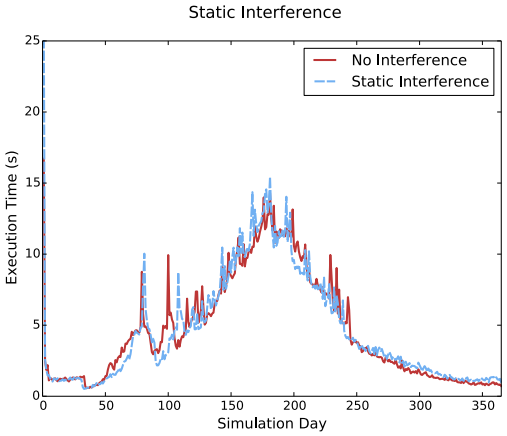

Figure 9 Execution time comparison with/without interference

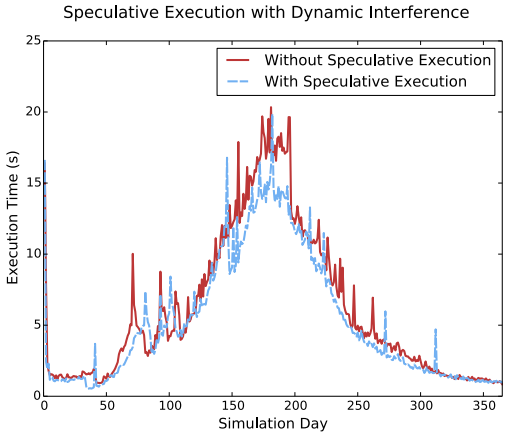

Figure 10 Execution time comparison with/without slow detection mechanism

depicted in Figure 9.

In the next experiment, we launched resource-intensive programs randomly during the execution of the simulation. The performance comparison with and without the speculative launch mechanism is shown in Figure 10. Under conditions of unpredictable interference, the total execution without the slowdown detection mechanism is 2216.7 seconds. With the slowdown detection mechanism, the total execution time is 1924.0 seconds, an improvement of 13.2%.

This experiment shows that even without the slowdown detection mechanism, the execution time is not affected greatly by interference in the cluster. This is due to the self-adjusting ability of the dynamic split and merge strategy: split and merge operations are based on predicted execution time, so the workload will be adjusted if a machine is slow or too heavily loaded. But examining the execution time in Figure 10 during peak execution times (day 150 to day 200), we see that the slowdown detection performs better overall. This is due the random timing of the launching of interfering programs. Split and merge operations are slow to react to changing circumstances because the operations are done only at the end of each simulation interval. The slowdown detection algorithm is much faster to react, since it uses frequent heartbeats to decide when to launch a speculative task.

We can conclude that the slowdown detection mechanism performs well in different environments with resource uncertainty. With or without interference from other users in the cluster, the slowdown detection mechanism outperforms the dynamic split and merge strategy without slowdown detection.

5. RELATED WORK

While our system primarily targets cloud or cluster deployments, checkpointing also plays an important role in creating fault-tolerant grid environments. Recognizing the importance of checkpointing schemes that dynamically adapt to their environment, [Chtepen et al. 2009] proposes a number of heuristics for determining a checkpoint interval. Unlike our system, their design does not require global state synchronization, but it does account for estimated execution time and checkpoint overhead to determine whether a job should be allowed to checkpoint its state or not.

Aurora [Park et al. 2006 and 2007] follows the controller-worker paradigm and also supports speculative tasks for fault tolerance and dealing with heterogeneity. Aurora shares many similarities with our framework, but does not feature a prediction-based fault tolerance module or deal with slowdown detection.

[Cucuzzo et al. 2007] proposes a heuristic-based method for autonomous checkpointing by each logical process in a distributed discrete event simulation. This technique accounts for simulation state information to determine when to checkpoint and in doing so completely avoids coordination overhead. However, this also results in situations where processes must “catch up” with the rest of the system if they have not recently recorded a checkpoint. Since our framework is designed for stochastic simulations, checkpointed state cannot deviate across simulation iterations.

[D’Angelo 2011] surveys the challenges and current trends seen in discrete event simulation parallelization and distribution, particularly noting the user-facing difficulties in running these simulations on a parallel architecture and how the trend toward cloud deployments has created a paradigm shift. For instance, distributed simulations that rely on optimistic parallelization with rollback suffer as network latencies increase; one possible solution may be conservative time synchronization [Vanmechelen et al. 2013]. The proposed multi-agent middleware solution is finer-

grained than our controller-worker model, but facilitating entity migrations would also require a more involved retrofitting process for sequential simulations.

In the context of distributed event execution with realtime requirements, [Feng and Lee 2007] propose an actor-based system that models interactions between components as dependencies. This allows the system to provide intelligent checkpointing and perform fine-grained rollback operations when a failure occurs, contrasting with our approach of synchronizing the simulation interval across all workers. However, this process does require more information about the events and their interactions.

Another checkpoint-based approach involving the structured high-level architecture (HLA) is presented in [Eklof et al. 2005]. In this case, checkpoints are stored in a universal stable storage repository by the federates, and several new publish-subscribe interactions are added to model checkpoint-related communications. As noted for the previously described technologies, this approach requires following a particular simulation structure a priori.

Ramirez Ortiz and Jiménez have also researched rollback operations and speculative execution in discrete event simulations. In their work, a coordinator handles snapshotting the simulation, but requires execution to completely halt while the process is carried out; our solution interleaves snapshot operations with other processing, and only requires the simulation to stop if a failure has occurred [Ramirez Ortiz and Jiménez 2011].

Lee et al. improves on standard slot scheduling algorithms in MapReduce by maintaining a pool of core resources and accelerator resources that are dynamically adjusted at runtime, similar to our "spare worker" concept [Lee et al. 2011]. In this case, the computing rate (CR) is calculated for each of the heterogeneous resources by running a pilot job on every possible hardware configuration. This helps identify machines that can benefit from GPU acceleration, have faster CPUs, etc., but is also a greedy approach that may cause overall throughput to decline in some situations. Based on this information, the *cloud driver* makes changes to the resource pool.

[Roy et al. 2011] uses predictive models for workload forecasting in the context of auto-scaling elastic clouds. Specifically, moving averages are computed over a sliding window to predict incoming loads and scale accordingly. In this case, both reconfiguration costs and SLA violations are taken into account when managing the virtual machines.

6. CONCLUSIONS AND FUTURE WORK

We have devised an adaptive checkpointing strategy and fault tolerance framework for discrete event simulations that dynamically selects both the timing and mechanism to perform checkpoints. The mechanism to perform checkpoints could either be full-fledged or deltas computed based on differences between successive full-fledged checkpoints. The decision is based on the time it takes to compute the checkpoint (and the overhead it adds to the overall simulation), the transmission overhead, and the amount of work that needs to be duplicated should the simulation be rolled back.

To our knowledge, this is the first attempt to incorporate fault tolerance into a discrete event simulation orchestrated using a stream-processing engine. All interactions and control messages are orchestrated as streams. Distributed workers that orchestrate this simulation run inside computations that are activated when data is available on their input streams.

Our framework can handle an arbitrary number of failures. The system can sustain permanent failures of all workers in the system. The system allows incorporation of redundancy for the checkpoint orchestrator (Speculator). With multiple Speculators, when the primary fails, one of the surviving speculators is promoted to be the primary. Where there is just one instance of the Speculator, we can sustain transient failures (such as restart of a machine) because Speculators manage their state by writing to, and reconstructing from, persistent local storage.

We have verified the correctness of our recovery strategy for the stochastic discrete event simulation. We have done this in the presence of multiple, concurrent failures and an adaptive checkpointing strategy.

The overheads introduced by our strategy are acceptable in situations where failures do not take place. In situations where a failure occurs, we reduce the recovery costs by minimizing the amount of work that needs to be duplicated.

6.1 Future Work

While the simple neural network we used to predict changes in system state performed well for our purposes, a Recurrent Neural Network (RNN) may provide better performance for making predictions due to their internal memory. In the future, we will evaluate different prediction techniques, including utilizing Elman networks.

Our delta checkpoint scheme provides different options to improve performance by allowing both the compression and suffix sorting algorithms to be changed at runtime. Selection of the appropriate algorithms could be performed automatically during the training process, or could be changed dynamically by the Speculator as conditions in the system change over time.

We also plan to adapt our parallelization and fault tolerance frameworks to oversee the execution of a different discrete event simulation to help identify common traits and execution patterns that affect fault tolerance functionality. This could incorporate new system health metrics and learning techniques to make our framework even more generalizable.

REFERENCES

A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. 2005. Hadoop: a framework for running applications on large clusters built of commodity hardware. http://hadoop.apache.org/.

Maria Chtepen, Filip HA Claeys, Bart Dhoedt, Filip De Turck, Piet Demeester, and Peter A Vanrolleghem. 2009. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. Parallel and Distributed Systems, IEEE Transactions on. Vol 20, No. 2 (2009), pp. 180-190.

William R Cotton, RA Pielke Sr, RL Walko, GE Liston, CJ Tremback, H Jiang, RL McAnelly, JY Harrington, ME Nicholls, GG Carrio, and others. 2003. RAMS 2001: Current status and future directions. Meteorology and Atmospheric Physics 82, 1-4 (2003), 5-29.

D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. Proceedings of the International Symposium on Distributed Simulation and Real-Time Applications, 2007, pp. 227–234.

G. D'Angelo. Parallel and distributed simulation from many cores to the public cloud. Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '11), 2011.

J. Dean and S. Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM 51, 1 (2008), 107-113.

L.P. Deutsch. 1996. DEFLATE compressed data format specification version 1.3.

M. Eklof, F. Moradi, and R. Ayani. A framework for fault tolerance in HLA-based distributed simulations. Proceedings of conference on Winter simulation, 2005, pp. 1182–1189.

K. Ericson and Shrideep Pallickara. 2012. On the performance of high dimensional data clustering and classification algorithms. Future Generation Computer Systems (2012).

K. Ericson, S. Pallickara, and C.W. Anderson. 2010. Analyzing Electroencephalograms Using Cloud Computing Techniques. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 185-192.

T. H. Feng, and E. A. Lee. Implementation of Real-Time Distributed Discrete-Event Execution with Fault Tolerance. UC Berkeley Tech Report. 2007.

C. Green and others. 2010. Simulation modeling of alternative control strategies for an HPAI outbreak using NAADSM. In Canadian Association of Veterinary Epidemiology Preventive Medicine (CAVEPM) Meeting, May 29 - 30 2010, Guelph, Ontario, Canada.

N. Harvey, et al. The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions. Preventive Veterinary Medicine 82, 3 (2007), 176-197.

Heaton Research, Inc. Encog Machine Learning Framework. http://www.heatonresearch.com/encog.

David Jefferson and James Leek. 2010. Application of parallel discrete event simulation to the Space Surveillance Network. In Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference, S. Ryan, ed.(Maui Economic Development Board, 2010) E, Vol. 34.

David Korn and K Vo. 2002. The VCDIFF generic differencing and compression data format. (2002).

Gunho Lee, Byung-Gon Chun, and Randy H. Katz. "Heterogeneity-aware resource allocation and scheduling in the cloud." Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud. Vol. 11. 2011.

J. MacDonald. 2008. XDelta. http://xdelta.org.

M. Malensek, S.L. Pallickara, and S. Pallickara. 2012. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals. Future Generation Computer Systems (2012).

M. Malensek, Z. Sui, N. Harvey, and S. Pallickara. 2013. Autonomous, Failure-resilient Orchestration of Distributed Discrete Event Simulations. Proceedings of the ACM Cloud and Autonomic Computing Conference. Miami, USA. 2013.

S. Pallickara, J. Ekanayake, and G. Fox. 2009. Granules: A lightweight, streaming runtime for cloud computing with support, for Map-Reduce. In Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on. IEEE, 1-10.

A. Park, and R. M. Fujimoto. Aurora: An Approach to High Throughput Parallel Simulation. Principles of Advanced and Distributed Simulation. PADS 2006. 20th Workshop on , vol., no., pp.3,10, 2006

A. Park, and R. Fujimoto. A scalable framework for parallel discrete event simulations on desktop grids, Grid Computing, 2007 8th IEEE/ACM International Conference on. Sept. 2007

David Patterson, Aaron Brown, Pete Broadwell, and others. 2002. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science.

D.L. Pendell, J. Leatherman, T.C. Schroeder, and G.S. Alward. 2007. The economic impacts of a foot-and-mouth disease outbreak: a regional analysis. Journal of Agricultural and Applied Economics 39, 0 (2007), 19-33.

Colin Percival. 2006. Matching with mismatches and assorted applications. Ph.D. Dissertation. University of Oxford.

K Portacci, A Reeves, B Corso, and M Salman. 2009. Evaluation of vaccination strategies for an outbreak of pseudorabies virus in US commercial swine using the NAADSM. In ISVEE 12: Proceedings of the 12th Symposium of the International Society for Veterinary Epidemiology and Economics, Durban, South Africa. 78.

Jorge Luis Ramírez Ortiz and Ricardo Marcelín Jiménez. 2011. Fault-tolerant Distributed Discrete Event Simulator Based on a P2P Architecture. In SIMUL 2011, The Third International Conference on Advances in System Simulation. 21-26.

N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. 2011 IEEE International Conference on Cloud Computing (CLOUD). July 2011.

K. Vanmechelen, S. De Munck, & J. Broeckhove (2013). Conservative distributed discrete-event simulation on the Amazon EC2 cloud: An evaluation of time synchronization protocol performance and cost efficiency. *Simulation Modelling Practice and Theory*, *34*, 126-143.

Volker Springel. 2005. The cosmological simulation code gadget-2. Monthly Notices of the Royal Astronomical Society 364, 4 (2005), 1105-1134.

Z. Sui, N. Harvey, and S. Pallickara. (2013). On the distributed orchestration of stochastic discrete event simulations. Concurrency and Computation: Practice and Experience.