Failure Recovery in Resilient X10*

DAVID GROVE, IBM T. J. Watson Research Center
SARA S. HAMOUDA†, Australian National University, Sorbonne Université, and INRIA Paris
BENJAMIN HERTA, IBM T. J. Watson Research Center
ARUN IYENGAR, IBM T. J. Watson Research Center
KIYOKUNI KAWACHIYA, IBM Research – Tokyo
JOSH MILTHORPE, Australian National University
VIJAY SARASWAT‡, Goldman Sachs
AVRAHAM SHINNAR, IBM T. J. Watson Research Center
MIKIO TAKEUCHI, IBM Research – Tokyo
OLIVIER TARDIEU, IBM T. J. Watson Research Center

Cloud computing has made the resources needed to execute large-scale in-memory distributed computations widely available. Specialized programming models, e.g., MapReduce, have emerged to offer transparent fault tolerance and fault recovery for specific computational patterns, but they sacrifice generality. In contrast, the Resilient X10 programming language adds *failure containment* and *failure awareness* to a general purpose, distributed programming language. A Resilient X10 application spans over a number of places. Its formal semantics precisely specify how it continues executing after a place failure. Thanks to failure awareness, the X10 programmer can in principle build redundancy into an application to recover from failures. In practice however, correctness is elusive as redundancy and recovery are often complex programming tasks.

This paper further develops Resilient X10 to shift the focus from failure awareness to failure recovery, from both a theoretical and a practical standpoint. We rigorously define the distinction between recoverable and catastrophic failures. We revisit the *happens-before invariance* principle and its implementation. We shift most of the burden of redundancy and recovery from the programmer to the runtime system and standard library. We make it easy to protect critical data from failure using resilient stores and harness elasticity—dynamic place creation—to persist not just the data but also its spatial distribution.

We demonstrate the flexibility and practical usefulness of Resilient X10 by building several representative high-performance in-memory parallel application kernels and frameworks. These codes are 10× to 25× larger than previous Resilient X10 benchmarks. For each application kernel, the average runtime overhead of resiliency is less than 7%. By comparing application kernels written in the Resilient X10 and Spark programming models

*This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published at https://doi.org/10.1145/3332372
†Research performed during PhD studies at the Australian National University
‡Research performed while employed at IBM T. J. Watson Research Center

Authors' addresses: David Grove, IBM T. J. Watson Research Center, groved@us.ibm.com; Sara S. Hamouda, Australian National University , Sorbonne Université , INRIA Paris, sara.hamouda@inria.fr; Benjamin Herta, IBM T. J. Watson Research Center, bherta@us.ibm.com; Arun Iyengar, IBM T. J. Watson Research Center, aruni@us.ibm.com; Kiyokuni Kawachiya, IBM Research – Tokyo, kawatiya@jp.ibm.com; Josh Milthorpe, Australian National University, josh.milthorpe@anu.edu.au; Vijay Saraswat, Goldman Sachs, vijay@saraswat.org; Avraham Shinnar, IBM T. J. Watson Research Center, shinnar@us.ibm.com; Mikio Takeuchi, IBM Research – Tokyo, mtake@jp.ibm.com; Olivier Tardieu, IBM T. J. Watson Research Center, tardieu@us.ibm.com.

0164-0925/2019/7-ART15 $15.00
https://doi.org/10.1145/3332372

we demonstrate that Resilient X10's more general programming model can enable significantly better application performance for resilient in-memory distributed computations.

ACM Reference Format:
David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. 2019. Failure Recovery in Resilient X10. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 15 (July 2019), 30 pages. https://doi.org/10.1145/3332372

1 INTRODUCTION

The explosive growth of compute, memory, and network capacity that is economically available in cloud computing infrastructures has begun to reshape the landscape of Big Data. The design and implementation of the initial wave of Big Data frameworks such as Google's MapReduce [Dean and Ghemawat 2004] and the open-source Hadoop system [Cutting and Baldeschwieler 2007; White 2009] were driven by the need to orchestrate mainly disk-based workflows across large clusters of unreliable and relatively low-performance nodes. Driven by increasing system capability and new compute and data intensive workloads, new programming models and frameworks have begun to emerge focusing on higher performance, in-memory distributed computing. Systems such as HaLoop [Bu et al. 2010] and M3R [Shinnar et al. 2012] enhanced the performance of MapReduce by enabling in-memory caching of data in iterative MapReduce workflows. Specialized systems such as Pregel [Malewicz et al. 2010], GraphLab [Low et al. 2012], MillWheel [Akidau et al. 2013], and many others were built to optimize the performance and programmability of specific application domains. More recently, the Apache Spark system [Zaharia et al. 2012] and its underlying Resilient Distributed Dataset (RDD) abstraction and data-parallel functional programming model have gained significant traction. The Spark programming model is significantly more general-purpose than prior Big Data frameworks. However, by design, Spark still presents a heavily restricted programming model. Spark focuses on functional data-parallel operations over immutable RDDs and declarative SQL-like operations over DataFrames [Armbrust et al. 2015]. Spark hides scheduling, distribution and communication decisions from the application programmer, and provides a single built-in approach to fault tolerance.

While transparent fault tolerance has obvious benefits, the one-size-fits-all approach has drawbacks too. Many applications can take advantage of domain-specific strategies for fault management that translate into all kinds of savings, e.g., time, memory, disk, network, power, etc. Some applications can evaluate or estimate the loss of precision resulting from a fault and decide to accept this loss. Scientific simulations can often rely on conservation laws—mass, volume—to fill gaps in data sets. The architecture of an application can also influence the choice of a fault tolerance approach. For instance, global checkpoints are well suited for bulk synchronous algorithms, whereas MapReduce workloads are better served by per-task checkpoints.

The Asynchronous Partitioned Global Address Space (APGAS) programming model [Saraswat et al. 2010] has been demonstrated to enable both scalable high performance [Milthorpe et al. 2015; Tardieu et al. 2014] and high productivity [Richards et al. 2014] on a variety of High Performance Computing (HPC) systems and distributed applications. Although originally developed in the context of the X10 language [Charles et al. 2005], the core concepts of the APGAS programming model can be found in a number of other HPC programming systems including Chapel [Chapel 2016], Habanero [Cavé et al. 2011; Kumar et al. 2014], Co-Array Fortran 2.0 [Yang et al. 2013], and UPC++ [Zheng et al. 2014]. Recent work on Resilient X10 [Crafa et al. 2014; Cunningham et al. 2014] enhanced APGAS with failure containment and failure awareness. An X10 application spans over a number of places, typically realized as separate operating system processes and distributed over a network of computers. When places fail, tasks running at surviving places continue to execute. Lost places and tasks are reported to survivors via software exceptions. Application programmers can implement exception handlers to react to place failures and take corrective actions. The order of execution of the surviving tasks cannot diverge from the failure-free execution, even in case of *orphan tasks*, i.e., tasks that have lost their parent task. This *happens-before invariance* principle is crucial to preclude races between orphan tasks and failure handling code. But it does not come for free as it requires the runtime system to maintain its control state using fault-tolerant algorithms and data structures.

Despite these advances, programming fault tolerance in Resilient X10 remains challenging. There is no built-in redundancy outside of the happens-before invariance implementation. Tasks at failed places cannot be respawned magically. Data at failed places is lost. Lost places are no longer available to host tasks or data, creating holes in the address space. In short, programming fault tolerance is rather difficult and error-prone. Moreover, there is little point to the exercise if the resilient code is significantly slower than the original. In most scenarios, running the non-resilient code repeatedly until success is a better trade-off. Beyond these practical concerns, there are also foundational issues. The formal failure model of Resilient X10 is too permissive: all the places can fail at once. The guarantees of Resilient X10 are formally valid in this scenario. But there is no way for an application to recover from such a catastrophic failure. While the Resilient X10 programmer can persist data by using an external data store, this is a priori a recipe for disaster as the happens-before invariance does not encompass foreign libraries.

In this paper, we revisit Resilient X10 to extend, improve, or revise aspects of the language, its semantics, and implementation to establish a practical general framework for efficient in-memory distributed computing with programmable fault tolerance. Our goal is to evolve Resilient X10 so that it not only enables failure recovery code to exist in theory, but makes the development of recovery code a *rewarding* experience. Our work is driven primarily by our experience in porting existing realistic applications, frameworks, and class libraries to Resilient X10 and in developing new applications. Our contributions provide dramatic increases to programmers' productivity and applications' performance:

- We rigorously specify resilient data stores and revise the failure model and happens-before invariance principle to accommodate them. We implement two resilient data stores with different trade-offs: a resilient store based on Hazelcast [Hazelcast, Inc. 2014], an off-the-shelf external distributed store, and a resilient store implemented in pure Resilient X10. With these stores, application programmers can trivially protect from failure application data deemed critical.
- We augment the language, its semantics, and runtime system to permit the dynamic creation of places. The combination of dynamic place creation with generalized indirect place addressing in the standard library enables *non-shrinking recovery*, that is, after recovery the program will have access to the same number of places as it did before the failure. This stability in the number of places significantly reduces the complexity of the application's failure recovery code since it avoids the need to redistribute data or otherwise change the program's communication topology.
- We identify and address performance bottlenecks in the existing open-source implementation of the happens-before invariance principle that cause up to 1000× slowdowns on common code patterns.
- We implement and empirically evaluate a suite of representative Resilient X10 application kernels including typical Big Data problems from the machine learning domain, scientific simulations, and global dynamic load balancing. Most are based on pre-existing X10 applications with small localized code changes for resiliency. These codes comprise a significantly more realistic corpus of APGAS programs—10× to 25× greater code size—than any prior

evaluation of Resilient X10. Across all our application kernels, the average overhead imposed by resiliency on non-failing runs was under 7%, and often well under.[1]
- Where possible, we compare the performance of the X10 kernels to equivalent kernels written using the Spark programming model to demonstrate that the additional flexibility provided by the APGAS programming model can yield significant performance benefits.

Section 2 presents the fundamental capabilities that the Resilient X10 system provides to the programmer; it includes a brief review of the APGAS programming model to provide necessary background. Section 3 illustrates how these capabilities can be combined to build resilient applications and frameworks. Section 4 describes key aspects of our implementation. Section 5 presents some of the application kernels we built to gain practical experience with Resilient X10 and provides an empirical evaluation of their performance. Finally, Section 6 covers additional related work and Section 7 concludes.

2 PROGRAMMING MODEL

This section presents an overview of the Resilient X10 programming model. The base X10 programming model (Section 2.1) and the semantics of resilient control (Section 2.4) are not new contributions of this paper. The failure model (Section 2.2) follows from prior work but is refined for this paper. Non-shrinking recovery (Section 2.3) and resilient stores (Section 2.5) are new contributions.

2.1 X10 Background

The X10 programming language [Charles et al. 2005] has been developed as a simple, clean, but powerful and practical programming model for scale-out computation. Its underlying programming model, the APGAS (Asynchronous Partitioned Global Address Space) programming model [Saraswat et al. 2010], is organized around the two notions of *places* and *asynchrony*.

Asynchrony is provided through a single block-structured control construct, **async** S. If S is a statement, then **async** S is a statement that executes S in a separate *task* (logical thread of control). Dually, **finish** S executes S, and waits for all tasks spawned (recursively) during the execution of S to terminate, before continuing. Exceptions escaping from S or tasks spawned by S are combined in a `MultipleExceptions` instance that is thrown by **finish** upon termination. Constructs are provided for unconditional (**atomic** S) and conditional (**when** (c) S) atomic execution.

A place is an abstraction of shared, mutable data and worker threads operating on the data, typically realized as an operating system process. A single APGAS computation may consist of hundreds or potentially tens of thousands of places. The construct **at** (p) S permits the current task to change its place of execution to p, execute S at p and return, leaving behind tasks that may have been spawned during the execution of S. The termination of these tasks is detected by the **finish** within which the **at** statement is executing. The object graphs reachable from the final variables used in S but defined outside S are serialized, transmitted to p, and de-serialized to reconstruct a binding environment in which S is executed. The snippet below shows how **finish**, **async**, and **at** can be combined to print a message from each place:

```
val msg = "Hello World";
finish for (p in Place.places())
  at (p) async
    Console.OUT.println(here+" says "+msg);
Console.OUT.println("GoodBye!");
```

[1] This number does not include the application-level checkpointing overhead, which can be decided arbitrarily and should reflect the expected mean time between failures (MTBF).

The messages from each place will be printed in an arbitrary order, but **finish** ensures they will appear before "GoodBye!" is printed.

Variables in one place can contain references (*global refs*) to objects at other places. Calling `GlobalRef(obj)` constructs a global ref to `obj` in the local heap. A global ref can only be dereferenced at the place of the target object.

Places are assigned numbers starting from zero. The application `main` method is invoked at place zero. The method `Place.places()` returns the set of places at the time of invocation; **here** evaluates to the current place.

2.2 Failure Model

Resilient X10 [Crafa et al. 2014; Cunningham et al. 2014] builds on X10 by exploiting the strong separation provided by places to provide a coherent semantics for execution in the presence of failures. It assumes a fail-stop failure model [Schlichting and Schneider 1983] where the unit of failure is the place.[2] A place `p` may fail at any time, with the instantaneous loss of its heap and tasks. The failure is *contained*: running tasks and heaps at other places are not affected by the failure of place `p`. In particular, if `q≠p`, any **at** `(q) S` initiated from place `p` or any other place before the failure of place `p` will execute to completion (see Section 2.4). Surviving tasks are made *aware* of failed places as follows. Any **at** `(p) S` executing at a place `q` will throw a `DeadPlaceException` (`DPE`). Any attempt to launch an **at** `(p) S` from place `q` will also throw a `DPE`. Global refs pointing to objects hosted at `p` now "dangle", but they cannot be dereferenced since an **at** `(p) S` will throw a `DPE`.

While this failure model makes it possible to reason about execution in the presence of failures, we need more to reason about failure recovery. Obviously an application cannot recover from a scenario where all places have failed at once, as there is no place left to run recovery code. In other words, not all failures can be recovered from. We have to draw a line between *catastrophic failures* and *recoverable failures*.

For this work, we extend Resilient X10 with the concept of a resilient data store. A resilient store is a safe haven for data (see Section 2.5). It is designed to transparently overcome place failures to avoid data loss. A store fails if and only if it loses data. The condition for a failure depends on the store implementation (see Section 4.2) and the actual content. For example, a store can be implemented to tolerate up to n concomitant place failures by maintaining replicas of each data element in $n+1$ places. A store can survive any number of infrequent failures over time if it rebuilds redundancy after each place failure. An empty store never fails. A place failure is defined to be catastrophic if it causes the failure of a resilient store instance.

Execution of an X10 program begins by executing the `main` method in a single task in place zero. As a result, X10 programs are typically structured with place zero containing a master task that coordinates overall execution. Therefore, Resilient X10 treats the failure of place zero as a catastrophic failure. This model is not unusual; for example Spark can recover from failed executors but a failure of the driver process (a Spark program's `main`) is a catastrophic failure. In Resilient X10 however, there is no requirement that place zero be a master place for all aspects of the execution, e.g., scheduling tasks, maintaining directories.

Our runtime and resilient store implementations do not assume that place zero cannot fail (see Section 4). While one of our implementations of the **finish** construct in Resilient X10 does make this assumption, we also offer a **finish** implementation that can survive the failure of place zero. Except for this special, opt-in implementation of **finish**, the runtime state and resilient data are replicated and distributed uniformly across all the places to protect from the failure of any place, including

[2]In a fail-stop failure model, the only failures are crash failures of servers. All non-crashed servers can detect that a crashed server has failed. Messages between servers are never lost unless either the sender or receiver crashes.

place zero, and ensure scalability. In principle, when used with the non-place-zero dependent **finish** implementation, our underlying runtime system could support running X10 as a service[3] where a failure of place zero is not considered catastrophic. However, we have not experimented with writing any applications that exploit this capability.

In summary, a place failure is catastrophic if and only if (i) the failed place is place zero or (ii) the place failure triggers the failure of a resilient store instance (data loss). In the remainder of this paper, we only consider recoverable, i.e., non-catastrophic, failures. Thanks to this definition, we can decompose the failure recovery problem into two independent subproblems: avoiding data loss by means of resilient data store (see Section 2.5 and Section 4.2) and preserving application behavior assuming no data loss (see Section 3 and Section 5).

2.3 Non-Shrinking Recovery

> *All problems in computer science can be solved by another level of indirection. – D. Wheeler*

Many APGAS applications contain structured data and structured communication patterns where places exchange specific data blobs with specific collections of other places. For example, row/column based broadcasts in distributed matrix operations or boundary data exchange with "neighbors" in a k-dimensional grid in scientific simulations. Prior work on Resilient X10 [Cunningham et al. 2014] only supported *shrinking* recovery. When a place fails, an application can reorganize to continue running with fewer places. However, for X10 applications with substantial distributed state, this reorganization often incurred a productivity and a performance cost. The programmer had to code the data movements explicitly and provide algorithms that work with flexible place counts. Often these algorithms would only imperfectly tolerate reduced place counts, resulting in imbalance that degraded future performance. To improve productivity and performance, we add to Resilient X10 support for *non-shrinking* recovery, i.e., the ability to compensate for lost places with fresh places, therefore greatly reducing the algorithmic burden for the programmer.

To permit non-shrinking recovery, we have augmented Resilient X10 with *elasticity*—the ability to dynamically add places to a running application. Elasticity is also useful by itself in cloud infrastructures where the availability and cost of resources vary dynamically. New places may be created externally, or may be requested internally by the running application via asynchronous invocations of `System.addPlaces(n)` or synchronous invocations of `System.addPlacesAndWait(n)`. After joining is complete, calls to `Place.places()` will reflect the new place(s). Numeric place ids are monotonically increasing and dead place ids are not reused. Higher-level abstractions, such as the `PlaceManager` described below, use these runtime calls internally to dynamically manage places, automatically compensating for lost places.

Because numeric place ids are managed by the runtime system and affected by place failures, they should not be directly targeted by application programmers. Instead, they should use X10 standard library abstractions such as `PlaceGroup` and `Team`. The `PlaceGroup` class represents an indexed sequence of places and provides methods for enumerating the member places and mapping between places and their ordinal numbers in the group. The `Team` class offers MPI-like collective operations. As a concrete example, a place `p`'s neighbors in a structured grid are usually computed as a simple mathematical function of `p`'s assigned grid id. Instead of using the place's actual numeric id, `p.id`, a Resilient X10 application should instead define a `PlaceGroup pg` containing all the constituent places of the grid and use `pg.indexOf(p)` as the grid id of `p`. In conjunction with the `PlaceManager` facility described below, consistent use of `PlaceGroup` indices in this way creates a level of indirection that is sufficient to enable the bulk of the application code to be used unchanged

[3] X10 as a service accepts and runs X10 tasks submitted to any place belonging the X10 service instance.

with non-shrinking recovery in Resilient X10. Many prior systems, including Bykov et al. [2011] and Chuang et al. [2013], have combined elasticity and logical naming to achieve similar high-level objectives.

The `PlaceManager` is a new addition to the X10 standard library that encapsulates place management for both shrinking and non-shrinking recovery. In essence, it implements a `PlaceGroup` that can be adjusted when a place fails. It exposes two primary APIs to higher-level frameworks and applications. First, it exposes an *active* `PlaceGroup`. Second, it has a `rebuildActivePlaces()` method that should be invoked when a place failure is detected to rebuild the active `PlaceGroup`. Depending on configuration, this method simply purges the dead places from the active `PlaceGroup`—for shrinking recovery—or replaces the dead places with fresh places—for non-shrinking recovery. The `PlaceManager` for non-shrinking recovery orchestrates the process of elastically requesting new places from the lower level X10 runtime system when necessary to replace dead places. It can be configured to keep an optional pool of "hot spare" places ready for immediate use. It uses hot spares if available (replenishing the pool asynchronously), or if none are available it waits for more places to be created. Finally, `rebuildActivePlaces()` returns a description of the places that were added/removed from the set of active places to enable application-level initialization of newly added places and updates.

While we could make the `PlaceManager` automatically react to place failures, in practice we observed that controlling the exact timing of the `rebuildActivePlaces()` invocation explicitly leads to cleaner code and simpler recovery logic than an implicit asynchronous invocation from the runtime system.

2.4 Resilient Control

X10 permits arbitrary nesting of **`async`**/**`at`**/**`finish`**. Hence when a place `p` fails it may be in the middle of running **`at`** `(q) S` statements at other (non-failed) places `q`. The key design decision in Resilient X10 is defining how to handle these "orphan" statements. While `S` has lost its parent place, it still belongs to enclosing **`finish`** and **`at`** constructs, e.g.,

```
finish { ... at(p) { ... at (q) S ... } } T
```

In a failure-free program, the execution of `S` happens before the execution of `T`. Resilient X10 maintains the strong invariant that *the failure of a place will not alter the happens-before relationship between statement instances at the non-failed places.* This guarantee permits the Resilient X10 programmer to write code secure in the knowledge that even if a place fails, changes to the heap at non-failed places will happen in the order specified by the original program as though no failure had occurred. Failure of a place `p` will cause loss of data and computation at `p` but will not affect the concurrency structure of the remaining code. In this example, if place `p` fails, `S` may execute or not depending on the timing of the failure. If `S` does execute, it will complete before `T` executes. Similarly, if place `q` fails, `S` may execute fully, partially, or not at all, but again (any surviving tasks spawned by) `S` will finish before `T` executes.

The operational semantics of X10 and Resilient X10, the happens-before relationship, and the invariance principle are formalized by Crafa et al. [2014]. Resilient X10 extends the base X10 semantics with transitions to model failures. The happens-before partial order is specified by means of execution traces: statement s_1 happens before s_2 if and only if s_1 occurs before s_2 in any trace containing s_2.[4] The invariance principle theorem states that if statement s_1 happens before statement s_2 viz. X10's semantics, then s_1 happens before statement s_2 viz. the semantics of Resilient X10.

[4]The happens-before relationship of [Crafa et al. 2014] relates activations of dynamic instances of statements in the execution of a given program and initial heap.

Cunningham et al. [2014] informally establish the link between the happens-before relationship and the control-flow constructs of the language. Moreover, they observe that although many constructs (sequence, conditional, loops, etc.) contribute to the partial order, only **finish** and **at** constrain the order of execution across places. In a sense, a sequence `S T` is naturally resilient as the loss of the ordering constraint cannot occur independently of the loss of `T`, which makes the constraint irrelevant. Therefore, only **finish** and **at** require new implementations for Resilient X10. We discuss our **finish** implementations, i.e., resilient distributed termination detection implementations in Section 4. The **at** construct is basically implemented as a **finish** with a single task.

Crafa et al. [2014] formalize X10's partitioned global address space, i.e., the distributed heap. The invariance principle therefore encompasses heap operations. In particular, a mutation of the heap is guaranteed to complete before any enclosing **finish** irrespective of any place shifts and place failures along the way. On the other hand, invocations of external services are not included in this formalization. While these invocations could be modeled as asynchronous tasks running at other places, we believe this would not make sense in practice. External services typically should not be expected to be aware of and contribute to (Resilient) X10's termination detection protocols. In particular, if a place fails just after invoking an external service, Resilient X10 cannot guarantee that a particular program statement will only happen after the completion of the invocation (whereas **finish** can offer this guarantee when dealing with invocations of asynchronous X10 tasks at failed places). But in practice, recovery code in Resilient X10 can still leverage the invariance principle to build strong ordering guarantees using mechanisms provided by the external service such as fences, epochs, transaction logs, etc.

2.5 Resilient Store

In order to enable applications to preserve data in spite of place failures, we extend Resilient X10 with the concept of a resilient data store realized as a distributed, concurrent key-value map. Since the APGAS programming model enforces strong locality—each object belongs to one specific place—a resilient data store is also partitioned across places. Invocations of the `set(key, value)` and `get(key)` methods of a resilient store associate a value to a key or return the value for a key for the current place. Map operations on a given key k at a given place p are linearizable.

Applications may use a resilient store to checkpoint intermediate results or sets of tasks (completed, in progress, pending). Upon failure, an application is responsible to replace or reconstruct the lost data using the content of the resilient store.

A resilient store ranges over an active `PlaceGroup` as defined in the previous section. In non-shrinking recovery, if a fresh place p replaces a dead place q, the map entries for place q are seamlessly transferred to place p. In short, the store content for place q survives the failure of place q and place p takes ownership of this content. For shrinking recovery, we support querying the content of the store of a dead place from any surviving place via the `getRemote(place, key)` method.

The resilient store implementations (see Section 4.2) handle the data replication and/or data movement needed to preserve the data. Using a resilient store is semantically equivalent to transferring objects across places, i.e., an object retrieved from the store is a deep copy of the object put into the store.

Resilient stores must obey the happens-before invariance principle (see Section 2.4). Store operations must happen in the order specified by the failure-free program. In particular, an update operation initiated from a task interrupted by the death of the hosting place must not linger. It must either mutate the store before any finish waiting for the task completes or never mutate the store. This property is crucial to ensure that recovery code can be constrained to happen after any store operations coming from the place whose death triggered execution of the recovery code.

A resilient store implementation in Resilient X10 can of course build upon **async** and **finish** to achieve happens-before invariance trivially. In contrast, integration of an off-the-shelf in-memory data grid in Resilient X10 may require some additional work to fulfill the requirement, such as flushing operation queues before reporting the death of a place to the X10 application.

3 BUILDING RESILIENT APPLICATIONS

This section illustrates how the core programming model concepts of Section 2 can be combined to define higher-level fault-tolerant application frameworks. We implement non-shrinking checkpoint/restart, a well-known technique for transparent fault tolerance in iterative applications. While Resilient X10 is intended to enable innovation in software fault tolerance, we want to devote this section to the programming model, not the particulars of an original or atypical fault tolerance algorithm. Moreover, we will use this algorithm as well as variations of this algorithm to bring fault tolerance to some of the application kernels presented in Section 5. We briefly discuss a few other approaches to resilience in Section 3.5.

3.1 Resilient Control

An iterative application typically looks like the following:

```
while(!app.isFinished()) app.step();
```

The `step` and `isFinished` methods, respectively, specify the loop body and termination condition. Each step may entail a distributed computation over the active place group of a `PlaceManager pm`.

Using Resilient X10, we can rewrite this loop to make it fault tolerant. The `execute` method below takes an instance of an `IterativeApp` and executes it resiliently, i.e., using checkpoint/restart to protect from place failures:

```
def execute(app:IterativeApp) {
 globalCheckpoint(app);
 var err:Boolean = false;
 var i:Long = 1;
 while(true) {
  try {
   finish {
    if(err) { globalRestore(app); i=1; err=false; }
    if(app.isFinished()) break;
    app.step();
    if(i % N == 0) globalCheckpoint(app);
    i++;
  }} catch(e:MultipleExceptions) {
   if(e.isDPE()) err = true; else throw e;
}}}
```

To invoke the `execute` method, the programmer must provide an instance of an `IterativeApp`, i.e., implement the methods listed in Figure 1. The code for `step` and `isFinished` is unchanged from the original non-fault-tolerant loop. The programmer must specify how to `checkpoint` and `restore` the *local* state of the application in between iterations. The `checkpoint` method should insert critical application data for the *current* place into a hash table. The `restore` method does the reverse. The programmer may also specify initialization code to run on dynamically created places by means of the `remake` method. Importantly, none of these methods need to handle data distribution or place

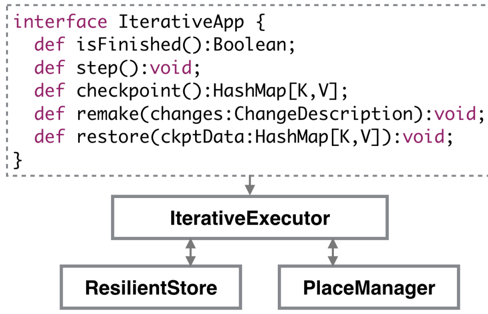

Fig. 1. X10 Resilient Iterative Framework

failures. The `globalCheckpoint` and `globalRestore` methods implemented in the next section orchestrate the invocations of `app.checkpoint`, `app.restore`, and `app.remake` to checkpoint and restore the global application state.

We now explain how fault tolerance is implemented by the `execute` method in details. The code first checkpoints the initial application state. The loop code cannot recover from a place failure before the completion of this first checkpoint. This invocation of `globalCheckpoint` is not in the scope of the try-catch construct. However, the application itself may be capable of replaying its initialization and invoke `execute` again.

The loop periodically makes checkpoints based on a configurable checkpointing interval `N`. It detects place failures and rolls back to the last checkpoint using a single exception handler. The handler distinguishes the dead place exceptions (using the `isDPE` helper method) that are transparently handled from other exceptions that abort the execution. The handler takes care of place failures at any stage of the loop, not only in `app.step` or `app.isFinished`, but also in `globalCheckpoint` and `globalRestore` using the same retry strategy for all failures. For instance, a place failure during the execution of `globalCheckpoint` sets `err` to **`true`**, which triggers the invocation of `globalRestore` when the while loop is reentered. The `globalCheckpoint` method implemented below uses double buffering to guard against incomplete checkpoints.

Together `execute`, `globalCheckpoint`, and `globalRestore` handle *any combination of non-catastrophic place failures* past the initial checkpoint. This includes not only failures during `app.step` or `app.isFinished`, but also during `globalCheckpoint` and `globalRestore`.

3.2 Resilient Data

The `globalCheckpoint` and `globalRestore` methods are implemented using the `PlaceManager` `pm` and a resilient store `rs`:

```
def globalCheckpoint(app:IterativeApp) {
 val k = key.equals("red") ? "black" : "red";
 finish for(p in pm.activePlaces()) at(p) async rs.set(k,
    app.checkpoint());
 key = k;
}
def globalRestore(app:IterativeApp) {
 val changes = pm.rebuildActivePlaces();
 rs.recover(changes);
 app.remake(changes);
```

```
  finish for(p in pm.activePlaces()) at(p) async app.restore(rs.get(key));
}
```

The two methods, respectively, invoke `app.checkpoint` and `app.restore` in every active place to extract the local state to checkpoint or restore it. Double buffering defends against failures during checkpointing. The checkpointing `key` is mutated only after finishing successfully all the local checkpoints. If any of the `app.checkpoint()` invocations fails, the control is transferred from the enclosing finish to the exception handler, skipping over the `key = k` assignment. Before attempting to restore the last checkpoint, the `globalRestore` method makes sure to rebuild the place group—replace dead places with fresh places—and reorganizes the resilient store accordingly. It also invokes `app.remake` to give the application the opportunity to process the changes, e.g., initialize data structures at the newly added places.

3.3 Discussion

At first, the fault tolerant loop code may seem daunting. After all, we started from one line of code and ended up with two dozen lines for `execute`, `globalCheckpoint`, and `globalRestore` combined. Most of the code however—the checkpointing interval logic, the error flag, the while loop, the invocations of `step`, `isFinished`, `globalCheckpoint`, and `globalRestore`—would be similar in any checkpoint/restart implementation. The logic is subtle but orthogonal to Resilient X10. The Resilient-X10-specific code follows a single pattern: the try-catch construct and the finish construct immediately inside of it. This pattern is enough to cover all non-catastrophic failure scenarios. Because it is so simple, it is easy to write, read, and maintain. In short, it is robust.

Moreover, the loop code in Resilient X10 can be refined or customized easily, whereas off-the-shelf checkpoint/restart frameworks typically offer a finite set of configuration flags or parameters. For instance, the initial checkpoint often has a broader scope than subsequent checkpoints because of immutable data (see Section 5). The input data may be reloaded or recomputed instead of checkpointed in memory. The X10 code can be adjusted to account for these variations. In contrast with off-the-shelf frameworks for transparent fault tolerance, Resilient X10 provides the means to tailor fault-tolerance schemes to specific workloads or application domains with benefits such as reduced performance overheads, reduced memory footprint, or improved recovery times. We discuss one such variant in the next section.

3.4 Resilient Iterative Executors

We added this checkpoint/restart framework to X10's standard library and used it to implement several application kernels discussed in Section 5. The `IterativeExecutor` class exposes an `execute` method that is essentially the same as the one presented here. We refer to this executor as a *global* executor; it can be used for algorithms that perform arbitrary communications as well as regular SPMD-style computations. For SPMD computations, the `step` method must start remote tasks at each active place, each task performing a single iteration. We implement an `SPMDIterativeExecutor` to better support this application pattern. This executor distributes the computation over the set of active places. It creates parallel remote tasks that run multiple iterations (up to the checkpointing interval) of the `isFinished` and `step` methods, which are no longer in charge of distributing the computation. By doing so, the SPMD executor eliminates the overhead of creating remote tasks at each step.

3.5 Other Approaches to Resilience

While bulk synchronous checkpoint/restart is one of the most commonly used techniques today, many applications can benefit from other approaches to resilience. Resilient X10 makes it possible to tailor recovery strategies to particular application domains or patterns.

In Section 5.2, we will demonstrate a resilient Unbalanced Tree Search application kernel that adopts a different approach. In this particular application, the tasks to be executed are highly unbalanced. Implementations that rely on periodic synchronizations or centralized schedulers perform poorly. The reference, non-resilient, state-of-the-art implementation uses distributed work-stealing to achieve high CPU utilization and low communication overheads. Using an iterative framework such as the one we have just described would require rearchitecting the application code and cripple performance. Therefore, we make independent checkpointing decisions in each place. A work transfer due to work stealing requires the synchronous (i.e., transactional) update of only two checkpoints, rather than updating them all in a bulk synchronous style. While the overhead per steal increases, the fundamentals of the scheduling scheme are preserved and the performance is good.

Some applications can estimate and possibly tolerate the loss of precision resulting from a fault. Scientific simulations can often rely on conservation laws—mass, volume—to fill gaps in data sets. For instance, a shallow water simulation that divides an area of interest into a grid and distributes grid elements across a compute cluster can reconstruct the water surface at a failed place using (i) the conservation of mass, (ii) the boundary condition, and (iii) a simple interpolation. Of course, the latter assumes the water is relatively calm. Resilient X10 makes it possible to not only implement such a recovery strategy but also dynamically switch between this strategy for water that is calm enough vs. a checkpoint-based strategy for water that is not.

4 IMPLEMENTATION HIGHLIGHTS

A feature of the X10 system is that a single X10 program can be compiled for execution on a wide variety of platforms and network transports with varying performance characteristics. X10 is implemented with two backends. On the *managed* backend, X10 compiles into Java and runs on (a cluster of) JVMs; on the *native* backend, X10 compiles into C++ and generates a native binary for execution on scale-out systems. X10's communication layer can use multiple underlying network transports including TCP/IP sockets, MPI, and PAMI. Resilient execution over MPI is supported using MPI User Level Failure Mitigation (ULFM) [Hamouda et al. 2016]. This diversity of implementation is valuable: different combinations are best suited for specific application domains or deployment scenarios. Therefore, our implementation of Resilient X10 includes both native and managed X10, three network transports (Java sockets, native sockets, MPI), and full support for Linux, Windows, and macOS.

The key implementation challenge in providing Resilient X10's happens-before invariant for resilient control is making X10's **finish** construct resilient. This entails adjusting the distributed termination algorithm used by **finish** to be failure aware and storing its distributed state in a (potentially specialized) resilient store. Logically, the resilient store used for **finish** is distinct from the resilient store used for application data. Three implementations of resilient finish were described in Cunningham et al. [2014]: one that stored all finish state at place zero, one that used ZooKeeper [Hunt et al. 2010] as an external resilient store, and one that used a custom resilient distributed store for finish state implemented in X10. The place zero approach is not scalable to large place counts. The use of a custom store was motivated by results showing that the ZooKeeper-based store was impractically slow, but the prototype custom store implementation could only survive a single place failure.

In this paper, we revisit the viability of using an off-the-shelf external distributed store. We implement resilient finish and the resilient store API on top of the Hazelcast in-memory data grid [Hazelcast, Inc. 2014]. Hazelcast offers a scalable, in-memory data grid which can be embedded in an X10 application to store control state as well as application data. For Resilient X10, we associate each compute node with a separate Hazelcast member. As data are backed up on multiple members, Hazelcast ensures that no data are lost in the event of a node failure.[5] We instantiate several Hazelcast distributed fault-tolerant maps to safeguard both the resilient application state and the runtime state of the resilient finish implementation. At this time, this implementation is only available with the managed backend.

We also continue to develop a pure X10 implementation of Resilient X10. We improved the place zero resilient finish performance. We developed a scalable resilient store in X10 that is capable of rebuilding redundancy on the fly, hence surviving multiple place failures. These artifacts are usable with both backends. In contrast to the Hazelcast implementation, the place zero finish cannot survive the failure of place zero. The resilient store, however, has no such limitation when instantiated in combination with Hazelcast finish.

The remainder of this section describes the major enhancements and extensions we have made over the system of Cunningham et al. [2014]. All have already been contributed back to the X10 open source project and were included in the X10 2.6.1 release [X10 v2.6.1 2017].

4.1 Resilient Control

All three prior implementations of resilient finish imposed a significant performance penalty on task creation. As a result, common X10 programming idioms that utilize fine-grained tasks would incur crippling overheads under Resilient X10 (see Table 1). This greatly reduced the practical usefulness of resilient finish by preventing the unmodified reuse of existing X10 frameworks and applications. We developed several optimizations to reduce the cost of task creation; they are presented below in order of their relative importance.

The most important problem to tackle was to minimize the resiliency imposed overheads on the very common operation of local task creation and local finishes. We did this by exploiting the insight that only a subset of the tasks actually need to be tracked resiliently to provide the full Resilient X10 semantics. In particular, the exact number and identity of tasks that were executing in a failed place is not observable in the surviving places. This insight allows a non-resilient place-local counter to be used to cheaply track the lifetime of each incoming task and its locally spawned descendants. The counter starts with a value of one to indicate the liveness of the already started incoming task; it is incremented when local children are spawned and decremented when tasks it is tracking complete. Interactions with a resilient store are only required when (i) a new remote task is spawned or (ii) when a local counter reaches zero, indicating termination of its local fragment of the task tree. Similarly, the existence of a finish does not need to be resiliently recorded until it (transitively) contains a non-local task. The combination of these two optimizations virtually eliminates the performance penalty of resiliency for fine-grained concurrency within a place.

Second, in non-resilient X10, spawning a remote task is mostly asynchronous: the parent task is not stalled waiting for the remote task to begin executing. More precisely, the parent task continues its own execution as soon as it has initiated the message send requesting the remote task creation and recorded the initiation of a remote task in the local portion of the distributed (non-resilient) state of its controlling finish. In all three original resilient finish implementations, spawning a remote task

[5]Earlier versions of Hazelcast promised strong consistency, however, starting from version 3.9 Hazelcast promises only eventual consistency, where a failure to correctly replicate a mutating operation is notified by throwing an exception. At the Resilient X10 layer, such an exception could be treated as a catastrophic failure of the resilient store.

entailed synchronous interactions with a resilient store. Synchronization with the store ensured that the termination of the parent task could not be observed by the resilient store before it observed the initiation of the remote child task. However, as shown in Table 1 below, the additional synchronization greatly increased the cost of fan out communication patterns. An important additional benefit of the local termination optimization described above is that it also provides a simple path to supporting asynchronous spawning of remote tasks that is independent of the implementation details of the resilient store. In effect, the X10 runtime system is enhanced to spawn an additional synthetic local child of the current task that is responsible for an asynchronous interaction with the resilient store. The presence of the additional local child allows the parent task to continue (and even terminate) without the possibility of its termination being prematurely reported to the resilient store resulting in incorrect early exit from the finish (the synthetic child ensures that the value of the local counter cannot reach zero until the communication with the resilient store has been completed). This recovers the mostly asynchronous spawning of remote tasks enjoyed by non-resilient X10.

Finally, an additional optimization can be applied to the place zero resilient finish to reduce the communication traffic during the spawning of a remote task. If the serialized data for the task is relatively small, the spawning place can send the task and data to place zero, which can update the resilient finish state and then transmit the task and data to the destination place (2 messages). The original protocol sent the task data only once directly from the source to destination places, but required a request/response interaction with place zero by both the source and destination places to update the resilient finish state (5 messages). We measured the performance of the place zero resilient finish with and without this optimization on the microbenchmark suite. For benchmarks that spawned remote tasks, it enabled performance gains ranging from 10% to 33%. It is important to note that this optimization is not generally applicable to distributed resilient stores because it relies on the strong invariant that place zero processes each message exactly once.

Table 1. Performance cost of resilient finish for important communication and concurrency patterns at small and medium scale. Each number is the slowdown vs. non-resilient finish to perform the same operation with the same number of places (1.0 indicates no slowdown).

	Slowdown factor vs. non-resilient finish							
Scenario	**PPoPP'14 place zero**		**Current place zero**		**PPoPP'14 distributed**		**Hazelcast**	
	8 places	80 places	8 places	80 places	8 places	*64 places*	8 places	80 places
Local work	945.4	909.9	1.1	1.1	*10.3*	*19*	1.0	1.0
Single remote activity	5.8	6.6	4.0	3.9	*5.8*	*5.5*	17.0	30.8
Fan out, message back	19.2	42.6	3.5	3.9	*6.4*	*4.2*	13.5	15.2
Fan out, local work	201.1	297.8	3.0	2.6	*5.9*	*4.8*	11.4	11.9
Fan out, fan out	9.0	192.9	4.8	2.0	*7.7*	*12.1*	10.4	1.2
Tree fan out	6.3	25.1	3.7	7.6	-	-	15.4	19.1

Using the microbenchmark suite from Figure 6 of Cunningham et al. [2014] as updated in the X10 2.6.1 release,[6] we studied the performance and scalability of resilient finish. Table 1 compares the performance of the PPoPP'14 resilient finish implementations as found in X10 2.4.1 (as cited in Cunningham et al. [2014]) to our current implementations. All the patterns use a single finish to manage the whole group of spawned tasks, except the tree fan out pattern which creates a binary tree of finishes each managing two remote tasks at two different places. The first and fourth rows demonstrate the effectiveness of our enhancements to eliminate resiliency overheads for purely local

[6]see x10.dist/samples/resiliency/BenchMicro.x10 from X10 v2.6.1 [2017]

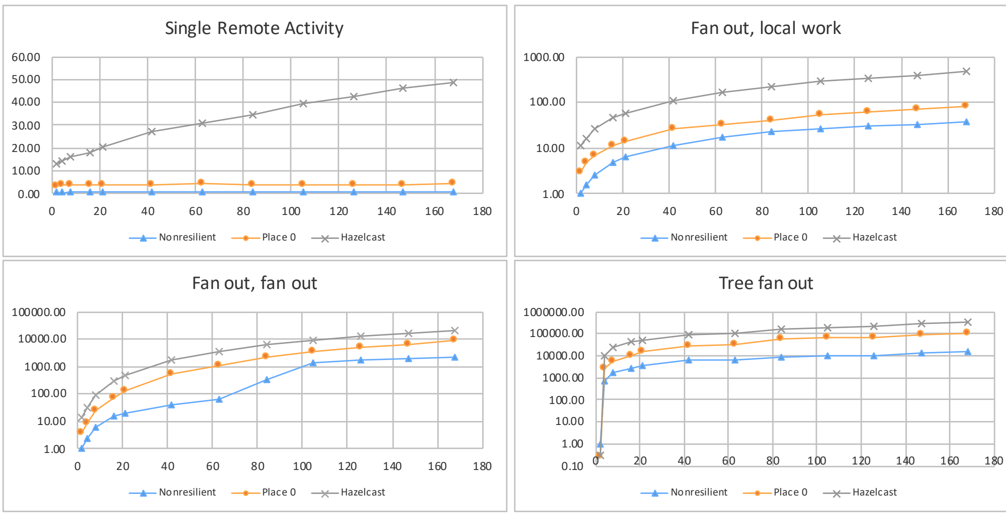

Fig. 2. Finish microbenchmarks with trivial task bodies. The graphs show the relative worst-case performance of our implementation of non-resilient, place zero, and Hazelcast finishes for four important concurrency/distribution patterns. The y-axis of each graph is the slowdown relative to non-resilient finish at 2 places; the x-axis is the number of places. For all but the top left graph, the amount of work increases with the number of places and the y-axis is logarithmic.

concurrency. Rows three through six show the impact of mostly asynchronous spawning of remote tasks. Overall the improvements to the place zero finish implementation are substantial, especially as the number of places increases.

Unfortunately, despite significant effort, we were unable to reproduce the PPoPP'14 distributed resilient finish results using the X10 2.4.1 release. All of the microbenchmarks containing remote activities failed to run correctly with X10's 2.4.1 distributed resilient finish implementation. Therefore the numbers in the PPoPP'14 distributed column of Table 1 are taken from the prior paper's raw experimental data. Comparing these columns with that of the Hazelcast-based resilient finish suggests that although there may be modest performance advantage to using a highly customized distributed store, using a general in-memory resilient store is a viable approach. The PPoPP'14 distributed resilient finish implementation was later removed in the X10 2.5.0 release, primarily due to lack of confidence in its correctness and maintainability.

Figure 2 shows the scaling graphs for our enhanced place zero and Hazelcast resilient finishes compared to non-resilient finish at 2 places. The scaling graphs provide a more detailed view than Table 1, which only presented data for 8 and 80 places. We expect there to be a cost to resiliency that depends on the implementation of resilient finish, the number of finish scopes executing concurrently, and the number of spawned remote tasks per finish. A truly distributed resilient finish implementation may have increasing overheads with an increasing number of spawned remote tasks, however, it is expected to provide better scalability than a place zero implementation in patterns that generate a large number of parallel finish scopes, such as the tree fan out pattern.

The top left graph shows the cost of spawning a single remote task. The message reduction optimizations for place zero finish enable overhead of less than 4× at all scales; overheads for Hazelcast increase from 13× to 49× as the number of places increases. The remaining three graphs represent commonly occurring APGAS work distribution patterns. The 'fan out' pattern (top right in Figure 2) is important as it is commonly used in X10 applications. The amount of termination detection work for a 'fan out' finish is O(N), where N is the number of places. A typical X10 program uses a 'fan out' finish multiple times, for example to assign work to each of the available places or to create global data structures that span all places. The 'fan out, fan out' pattern (bottom left in Figure 2) creates a single direct task to each of N places, each of which in turn creates N tasks to all places. Therefore, the expected complexity of termination detection is O(N^2). The 'tree fan out' pattern (bottom right in Figure 2) creates a binary tree of tasks at N places, with a complexity of O(log(N)). Figure 2 shows that, for the three patterns, place zero stays within 10× of the non-resilient finish and Hazelcast within an additional 2× to 5× of the place zero finish. While these numbers remain high in the absolute, our experimental study demonstrates that they are now good enough to support the programming model in practice.[7] The overhead of resiliency including resilient finish but excluding application-level checkpointing remains below 7% for all applications considered (see Section 5). We are currently implementing a native distributed finish implementation that is expected to outperform the Hazelcast implementation and deliver better scalability for task decomposition patterns that create a large number of parallel finish scopes.

4.2 Resilient Stores

We experimented with a number of approaches and decided to focus on two implementations: a resilient store based on Hazelcast and a resilient store implemented in X10.

We provide a common store API, so that the store implementation can be decided at application startup time. The core API consists of the `get(key)`, `set(key, value)`, and `getRemote(place, key)` methods discussed in Section 2.5.

4.2.1 Hazelcast-based store. This store is implemented using a distributed Hazelcast map. The resilient store `get` and `set` methods are mapped to Hazelcast's homonymous methods by appending the place index in the active place group to the key. Method `getRemote` also simply maps to Hazelcast's `get` method.

Catastrophic failures depend on the Hazelcast configuration. In our experiments, we configure Hazelcast with one synchronous backup, i.e., one level of redundancy. The store can survive multiple place failures as long as the failures are distant enough in time for Hazelcast to rebuild its redundancy in-between failures.

4.2.2 X10 Resilient Store. We implement a resilient store in X10 by maintaining two replicas of the data. The key value pairs at place `p` (master) are transparently replicated at the next place in the active place group (slave). Store read operations only access the master replica (local). Write operations require updating both the master and the slave as follows:

```
finish at (slave) async slaveStore.set(key, value);
masterStore.set(key, value);
```

The resilient finish ensures the slave is updated successfully before the master, thus guaranteeing that no value can be read from the store before being replicated. If the slave dies before or during the update, the write fails with a `DPE`. A lock (not represented) ensures no two writes can overlap.

The store is constructed over the set of active places in a `PlaceManager`. It has a method `recover(changes)` that should be invoked when a process failure is detected. The `changes` parameter is obtained from the `PlaceManager`; it includes the new set of active places, as well as the set of added/removed places since the last invocation for the PlaceManager's `updateActivePlaces()`

[7] As described in more detail by Tardieu et al. [2014], the `PlaceGroup` class in the X10 standard library provides convenience methods that compensate for the O(N) complexity of finish by implementing a scalable 'fan out' communication pattern with a dynamically constructed tree of **finish** instances. Since they are simply compositions of **finish**, these highly scalable `PlaceGroup` methods are also available in Resilient X10.

method. The store replaces each removed place with an added place at the same ordinal location. Each removed place had previously held a master replica for its own data, and a slave replica for its left neighbor. These replicas are now lost, however, copies of them are available at other places, assuming no catastrophic failure happened that caused the loss of two consecutive active places. The copies are fetched. They provide the initial state of the store at the fresh places.

Like the Hazelcast store, this store can survive any number of place failures, provided failures happen one at time, with enough time in-between for the store to rebuild the lost replicas. The store is implemented with less than 500 lines of X10 code, and can be considered an application study in its own right which demonstrates the expressiveness of the Resilient X10 model. It supports a much richer API than the core API we discuss in this paper. In particular, it handles local transactions, where multiple keys are accessed atomically at the same place. A local transaction object, e.g. `tx`, can be created at the master replica by calling `startLocalTransaction`. An activity can submit a group of get and set operations to the store through the `tx` object, by calling `tx.get(key)` and `tx.set(key, value)` methods and commits the transaction by calling `tx.commit()`. The execution of concurrent local transactions at the same place can result in conflicts if two transactions are accessing the same key and at least one of them is writing. We currently avoid this scenario by executing the transactions in order, however, more sophisticated concurrency control mechanisms are also feasible to implement. During transaction execution, write operations are performed on shadow copies of the data at the master replica. A transaction log records the updated keys and their new values. At commit time, the transaction log is applied at the master replica only after successfully updating the slave. A failed slave results in aborting the transaction at the master replica by discarding the log and throwing a `DeadPlaceException`.

4.2.3 Distributed Transactions. One of our applications (see Section 5.2) requires the ability to atomically update the local store and a remote store. The application is such that no conflicting updates can ever occur. The X10 resilient store currently lacks support for distributed transactions. To support this application, we implement the method `set2(key1, value1, place2, key2, value2)` using a simple transaction log. The transactions in progress (logged) are replayed after a place failure, before accessing the store to restore the application state. The log itself is also implemented as a resilient store.

4.3 Elasticity

Enabling elasticity required enhancements to all levels of the X10 implementation stack: the launching infrastructure that creates the initial processes, the network transports that bind them together, the core runtime that implements the PGAS abstractions, and a variety of standard library classes that are built on top of the PGAS abstractions. Additionally, in a cloud environment, acquiring the necessary computational resources to execute the additional processes that will become the new places requires negotiation with cluster management software.

Our current implementation fully supports elasticity for Managed X10 including an integration with the Apache Hadoop YARN [Vavilapalli et al. 2013] cluster resource manager. With a single additional command line argument, Managed X10 applications can be launched on a YARN-managed cluster and the implementation of `System.addPlaces(n)` will automatically acquire new containers from YARN and launch the new places within them.

Although much of the runtime implementation is shared by Managed and Native X10, elasticity support for Native X10 is not yet complete. The primary gap is at the X10RT network layer: none of Native X10's X10RT implementations support the dynamic addition of new places after initial application launch. Adding such support to Native X10's TCP/IP-based `x10rt_sockets` transport could be done with modest development effort.

5 APPLICATION STUDIES

We developed a number of resilient application kernels to assess the flexibility of the Resilient X10 programming model and the capabilities of our implementations. Most codes are derived from existing X10 kernels and frameworks that were extended to make them resilient. All of our enhancements to the X10 runtime have already been incorporated into the master branch of the main X10 git repository [The X10 Language 2019], and have been part of the X10 2.6.1 release.

This section presents four such resilient application kernels—Unbalanced Tree Search, KMeans, PageRank, LULESH—chosen to illustrate how different aspects of the programming model can be combined to achieve flexible resiliency solutions that best meet application needs. Each subsection describes the kernel, the design decisions made to make it resilient, and experimental results including direct comparisons with Spark-based implementations for the first three kernels.

5.1 Experimental Setup

All experiments were conducted on a 23-node compute cluster. Each node contains two quad-core AMD Opteron 2356 processors and 12 GiB-16 GiB memory. The cluster is connected via a 4×DDR Infiniband network using IP over IB. The compute nodes run RHEL 6.9 and the cluster is managed using Apache YARN 2.6.0. For comparisons with Spark, we used Apache Spark 2.0.1 with `-master yarn`. Our X10 implementation is a pre-release version of X10 2.6.1, the most recent open source release of X10. The JVM for both Managed X10 and Spark was Oracle Java HotSpot Server version 1.8.0_101.

For each application, we are primarily interested in three scenarios: non-resilient execution, failure-free resilient execution, and resilient execution with three place failures during a single run. Application parameters were chosen to achieve runs lasting approximately five minutes. This gives sufficient time to amortize application and JVM warmup while being short enough to permit a large number of runs to be completed. We inject failures by killing processes with a timer to guarantee that there is no correlation between the failure time and the ongoing computation. Failures are spaced by at least 30s to ensure no catastrophic failure occurs. Of course, this failure scenario is unrealistic. Mean time between failures (MTBF) is typically much longer. Our experimental protocol is intended to stress the runtime system and demonstrate its reliability more effectively than a single-failure scenario would.

For Resilient Managed X10, we use Hazelcast version 3.7.1 as the underlying store for both resilient finish and the resilient data store. This represents a scalable solution based on a production-level fully-distributed store. In the three-failure scenario, Resilient Managed X10 is configured to maintain one “hot spare” place; the `PlaceManager` will asynchronously replace the spare place after each failure to minimize future recovery time. As the Hazelcast-based resilient finish and resilient store are only implemented for Managed X10, for Resilient Native X10 we use the place zero resilient finish and the X10 resilient store of Section 4.2.2. Because Native X10 does not support elasticity, the three-failure scenario requires starting with three spare places. Therefore, unless otherwise noted, all experiments use 20 nodes (160 cores) for application execution. For X10, this corresponds to 20 active X10 places, each with X10_NTHREADS=8. For Spark it corresponds to 20 executors, each with 8 cores. This enables apples-to-apples comparison of application throughput across all configurations.

Unless otherwise stated, all execution times are the mean of at least 15 runs and the 95% confidence intervals are less than 1% of the computed averages for X10. Spark performance on 15 runs is less predictable with 95% confidence intervals ranging from 1% to 7% of the mean.

5.2 Global Load Balancing: UTS

Lifeline-based global load balancing (GLB [Saraswat et al. 2011; Zhang et al. 2014]) is a technique for scheduling large irregular workloads over distributed memory systems. The Unbalanced Tree Search benchmark (UTS [Olivier et al. 2007]) is the canonical benchmark for GLB. An X10 implementation of UTS using the GLB approach has been shown to scale to petaflops systems with tens of thousands of cores [Tardieu et al. 2014]. Our baseline UTS implementation is similar but uses multiple threads/workers per place so we can fully utilize a node with a single place. It is only intended for the managed backend as it uses Java's MessageDigest API for computing cryptographic hashes.

UTS measures the rate of traversal of a tree generated on the fly using a splittable random number generator. A sequential implementation of UTS maintains a queue of pending tree nodes to visit initialized with the root node. It repeatedly pops a node from the queue, computes and pushes back the children ids if any, until the queue is empty.

The distributed implementation divides this queue among many worker threads by dynamically migrating node ids from busy workers to idle workers using a combination of stealing and dealing. There is no central scheduler. An idle worker can request work from a random peer. The code has a simple structure. At the top a finish waits for all the workers to terminate. Requests and responses are implemented with remote tasks. There is more to the load balancing than random work stealing, but this does not fundamentally affect the fault tolerance problem.

To add resilience to UTS,[8] the workers checkpoint their progress to a resilient store. Each worker stores how many nodes it processed so far, as well as the node ids in its queue. The lack of a central scheduler and global synchronization is important for the performance of the non-resilient algorithm. We want to preserve this property in the resilient code. Therefore workers independently decide when to checkpoint based on individual progress and idleness. Before sending work to an idle worker, the sender updates the checkpoints of both the sender and the receiver in one transaction (see Section 4.2.3). While the collection of checkpoints is constantly changing and may never reflect the progress of all workers at one specific point in time, it is always correct, i.e., the aggregated node count is consistent with the aggregated pending node lists. Upon place failure, all workers abort (possibly doing a last checkpoint) and fresh workers load the checkpoint and resume the traversal. The dominant task pattern in UTS is the fan out finish, which is used for initializing the places, performing the computation-intensive tree generation task at each place, and collecting the number of traversed nodes by all the places for computing the tree traversal rate. Checkpointing and work-stealing are performed concurrently at each place using finish constructs that create a maximum of one remote task. With the distributed Hazelcast store, concurrent handling of these small finishes has less significant impact on the scalability of the application, therefore, the expected scalability model for Resilient UTS is O(N), where N is the number of active places.

For comparison purposes, we have implemented UTS in Spark using a map/reduce strategy. The tree traversal is divided into rounds. In each round the global pending node list is split into p fragments producing to p independent tasks that can be scheduled in parallel. Each task traverses up to n tree nodes before returning the updated node counts and lists to the global scheduler. We tuned p and n to achieve the best performance for our benchmark configuration.

Evaluation. Table 2 compares the execution time and the rate of traversal expressed in million nodes per second of the sequential X10 code, the distributed non-resilient code, the resilient code without and with three place failures, and the Spark code. We run with managed X10. At scale, we

[8] The code for Resilient UTS is in the `ResilientUTS` directory of the benchmarks repository at [X10 Benchmarks 2019].

Table 2. UTS execution times (seconds) and throughput (Mnodes/s) using Managed X10 and Spark

	Depth	Time	Throughput
Sequential X10	14	164.8	6.43
Non-resilient X10	18	267.7	1011.3
Resilient X10 + Ckpt	18	268.4	1008.4
Resilient X10 + Ckpt + 3 Failures	18	277.3	976.1
Spark	18	376.8	718.8

use a tree of about 270M nodes (fixed geometric law with seed 19 and depth 18). For the sequential code, we reduce the depth to 14. The throughput of the sequential code does not depend on the depth.

The sequential code achieves 6.43Mnodes/s in average. The distributed code, with 160 cores, achieves 98% of the sequential code efficiency. Adding fault-tolerance adds less than 1% overhead. Each place lost reduces throughput by about 1.1%. The failure-free resilient execution takes 268.4s in average. Each loss increases execution time by about 3s. Roughly half of the 3s is taken to detect the place failure and recover: updating the active place group and initializing the workers. We attribute the other half to lost work, startup cost, and the cost of rebuilding redundancy. While the spare place pool mitigates the startup latency, the fresh JVMs have not been trained to run the UTS code. Hazelcast rebuilds the resilient map redundancy in a few seconds taking resources away from the tree traversal and increasing the latency of the resilient store. Without a spare place pool, the recovery time increases to 14s per failure.

In comparison, the Spark implementation only achieves about 70% of the efficiency of the sequential X10 code (without node failures). This is not surprising. We observe that the generated tasks complete in anywhere between a few tens of milliseconds to a few seconds leading to a lot of imbalance. Overdecomposition does not improve this result.

5.3 KMeans Clustering

KMeans clustering is a commonly used kernel for unsupervised learning. We implement a distributed version of Lloyd's iterative algorithm [Lloyd 1982] in X10. Our base implementation contains 220 lines of code. Implementing checkpoint/restore, adding resiliency testing scaffolding, and conforming to the `IterativeApp` interface of the global resilient executor framework of Section 3.4 required modifying 16 existing lines of code and adding 57 new lines. We use KMeans to demonstrate how the Resilient X10 programming model supports application kernels with substantial immutable distributed data structures (the input data) and modestly sized but rapidly changing mutable data (the current estimate of the cluster centroids). Thus, the initial checkpoint must persist GBs of input data while subsequent checkpoints need only ensure that the current estimate of the cluster centroids can be recovered. In fact, because the current cluster centroids are broadcast to every active place at the start of each iteration, it is not necessary to actually checkpoint the centroids. Upon failure, they can be recovered from any surviving place and the computation can continue with at most the loss of one iteration of work. Therefore in our X10 implementation,[9] after the initial checkpointing of their input data, the active places do not actually store any state in response to checkpointing requests from the iterative framework. KMeans is entirely implemented as a series of fan out finish blocks with local work at each place, therefore, its expected scalability model is O(N), where N is the number of active places.

[9]see `x10.dist/samples/resiliency/ResilientKMeans.x10` in the X10 git repository [The X10 Language 2019]

For comparison we use two variants of the KMeans implementation from Spark's MLLib. The first is the unchanged MLLib code, which is capable of handling input data containing both sparse or dense vectors. The second is our manual specialization of the MLLib implementation to only handle dense vectors, which is a fairer comparison to our X10 implementation. For both Spark variants we persisted the RDD containing the input data with `StorageLevel.MEMORY_ONLY_2` to match X10's in-memory persistence strategy for this data.

Table 3. KMeans execution times (seconds)

	Total Time	Single Step
Managed X10	283.4	5.64
Resilient Managed X10	291.5	5.79
Resilient Managed X10 + Ckpt	318.7	5.79
Resilient Managed X10 + Ckpt + 3 Failures	389.5	5.90
Native X10	195.9	3.90
Resilient Native X10	196.1	3.91
Resilient Native X10 + Ckpt	199.4	3.91
Resilient Native X10 + Ckpt + 3 Failures	229.9	3.90
Spark MLLib	473.6	8.92
Spark DenseVector	368.2	6.81

Evaluation. Table 3 shows the total execution times[10] and single step times for 50 steps of the KMeans algorithm configured to find 300 clusters over an input of 20,000,000 30-dimensional points represented as dense vectors. When checkpointing is enabled, the initial checkpoint averaged 21.8 seconds for Resilient Managed X10 and 3.1 seconds for Resilient Native X10. Spark averaged 27 seconds to persist the input RDD. Checkpointing time accounts for 27.2 of the 35.3 second gap between Managed X10 and Resilient Managed X10 + Ckpt. Runtime overheads, primarily that of the Hazelcast-based resilient finish, account for the remaining 8.1 seconds (less than 3%) of overhead.

These results also illustrate the advantage of Native X10 for numerically intensive loop-based kernels: it significantly outperforms Managed X10, which in turn outperforms Spark. This performance difference is primarily attributable to the effectiveness of the underlying compilers in generating efficient machine code for the computationally intense loop nest that is the heart of the KMeans computation. The exact same KMeans X10 code is more effectively optimized when it is compiled to C++ and statically compiled by the platform C++ compiler than when it is compiled to Java and JIT compiled by the JVM. Similarly, the JVM's JIT compiler is able to do a better job optimizing the bytecodes generated from the X10 version of the key loop nest than it does for those generated from Spark's Scala version of the loop.

On the runs with three failures, there is an average 70.8 second (23.6 per failure) performance drop for Resilient Managed X10. As with UTS, approximately 2 seconds can be attributed to failure detection and recovering the X10 runtime system. Restoring the application state from a checkpoint averages 13 seconds per failure. We attribute the remaining 9 seconds to lost work (50% of an iteration is 3 seconds) and JVM warmup of the newly added place (which takes 3-5 iterations to

[10]For KMeans, the 95% confidence interval for Resilient Managed X10 is 1.5% of the mean and 3.5% for Resilient Managed X10 with failures.

reach peak performance). Since KMeans is an SPMD-style algorithm, performance is gated by the slowest place.

5.4 Global Matrix Library: PageRank

The X10 Global Matrix Library (GML) implements distributed linear algebra operations over matrices in a variety of dense and sparse formats [Hamouda et al. 2015]. It includes a set of benchmark codes using common algorithms for data analytics and machine learning. The core GML library consists of 20,500 lines of X10, 2,100 lines of C++ and 250 lines of Java. To support resilience in GML, snapshot and restore methods were implemented for the key matrix and vector classes.

We evaluate the cost of resilience for the GML PageRank benchmark[11] using the SPMD resilient executor described in Section 3.4. Approximately 50 lines of codes were added or modified from the original implementation to conform with the `IterativeApp` interface. In contrast, Cunningham et al. [2014] were not able to base their resilient SpMV kernel on the existing GML code base; they wrote 536 lines of new custom code. We compare with the Spark/GraphX [Xin et al. 2014] PageRank SynthBenchmark implementation. The expected scalability model of the GML PageRank benchmark is O(N), where N is the number of active places. It uses the fan out finish pattern, with complexity of O(N), multiple times for constructing distributed matrices, initializing input data, and starting an activity at each place to execute the steps of the PageRank algorithm. The steps use collective operations from the `Team` class, which organizes the places in a binary tree structure and has a complexity of O(log(N)).

Table 4. PageRank execution times (seconds)

	Total Time	Single Step
Managed X10	292.6	9.75
Resilient Managed X10	312.9	10.4
Resilient Managed X10 + Ckpt	440.8	10.4
Resilient Managed X10 + Ckpt + 3 Failures	684.1	14.9
Spark/GraphX	996.8	33.2

Evaluation. We measured the time to compute 30 iterations of PageRank for a randomized link matrix with 5 million pages and 633 million edges using a log-normal distribution of edges with $\mu = 4$ and $\sigma = 1.3$ as per Malewicz et al. [2010]. For Spark/GraphX, the number of edge partitions `numEParts` was set to twice the total number of cores.

Table 4 shows the total time and time per iteration.[12] The first checkpoint for PageRank is very slow at 82.0s, as it includes the immutable link matrix (about 10GiB for this problem size). Subsequent checkpoints are much faster at 5.1s as they only store the mutable PageRank vector (40MiB). Excluding the checkpointing time, the overhead of resiliency is less than 7% over the non-resilient execution time.

Using a checkpoint time of 5.1s, we used Young's formula to approximate the optimum checkpoint interval for each problem size: $\sqrt{2 \times t_{\text{checkpoint}} \times \text{MTBF}}$, where MTBF is the mean time to failure [Young 1974]. Assuming a high failure probability—MTBF of 60 seconds for the full cluster—the optimum checkpoint interval is 24.7s or approximately 3 iterations.

[11] The code for PageRank and the GML framework it uses are in the main X10 git repository [The X10 Language 2019] in the directories `x10.gml/src` and `x10.gml/examples/pagerank`, respectively.

[12] For PageRank, the 95% confidence interval is 1.6% of the mean for Resilient X10 and 2.9% for Resilient X10 with failures.

Resilient Managed X10 is around 2.3× faster than Spark/GraphX. For comparison, Kabiljo et al. [2016] report an Apache Giraph implementation of PageRank is 2× to 4× faster than Spark/GraphX (for a large Twitter graph).

On the runs with three failures, there is an average 243.3 second performance drop for Resilient Managed X10 (81.1s per failure). Approximately 2s per failure can be attributed to failure detection and recovering the X10 runtime system. Restoring the application-level state from a checkpoint averages 31.8s per failure. Another 21s is attributable to the loss of an average of two iterations per failure. We conjecture the significant slowdown of the average iteration time results from the combination of a cold JVM—GML PageRank is a much larger body of code than, say, KMeans—and the overhead of the memory management associated with the large amount of resilient data. Even with 3 failures, Resilient Managed X10 remains around 30% faster than Spark/GraphX running with no failures.

5.5 Scientific Simulations: LULESH

The LULESH proxy application [Karlin et al. 2013] simulates shock hydrodynamics on an unstructured mesh. Each place holds a rectangular block of elements, as well as the nodes that define those elements. Like the previous applications, the fan out finish pattern is used for creating distributed data structures and an activity at each place to execute local work for each step of the application. At each time step, a series of stencil operations are applied to update node-centered kinematic variables and element-centered thermodynamic variables. As the stencil operations require element- or node-centered values from a local neighborhood, it is necessary to exchange boundary or ghost regions between neighboring processes. The ghost region exchange is implemented between neighbors using global references to pre-arranged communication buffers and pair-wise synchronized one-sided get and put operations. LULESH also includes a spectrum of intra-place concurrent loops that rely on local **`finish`**/**`async`** patterns. Each iteration, all places agree on an adaptive time step using a collective allreduce operation. The X10 implementation of LULESH exploits both intra- and inter-node parallelism, and is around 10% faster than the reference implementation using C++/OpenMP/MPI across a range from 125 to 4,096 places (750 to 24,576 cores) [Milthorpe et al. 2015].

We modified LULESH[13] to more abstractly specify its communication patterns using `PlaceGroups` over subsets of active places, to use the SPMD resilient executor described in Section 3.4, and to add support for checkpoint/restore of all of its per-place data structures. LULESH contains approximately 4,100 lines of code; supporting resiliency entailed adding 106 new lines and modifying 94 other lines. Our LULESH code is a significantly more realistic example of a scientific simulation than the 175 line Heat Transfer kernel used in Cunningham et al. [2014].

The overhead of the fan out finish pattern, $O(N)$, is expected to dominate the overhead of the parallel finish blocks used in exchanging ghost cells, $O(1)$, and collective operations, $O(\log(N))$, therefore, LULESH's scalability model is $O(N)$, where N is the number of active places. However, using the place zero finish implementation is expected to cause a performance bottleneck for LULESH at large scales and result in high resilience overhead. We are currently implementing a native distributed finish implementation, which aims to address this limitation in LULESH and similar applications.

Evaluation. Table 5 shows the execution time in seconds using Native X10. We do not report times for LULESH on Managed X10 because LULESH heavily relies on stack allocation of worker-local temporary arrays for performance in its parallel for loops. Since Managed X10 does not support this Native X10 feature, LULESH performs quite poorly on it.

[13]The code of LULESH is found in the `lulesh2_resilient` directory of the applications repository at [X10 Applications 2019].

Table 5. LULESH execution times (seconds)

	Total Time	Single Step
Native X10	210.2	0.0875
Resilient Native X10	210.6	0.0875
Resilient Native X10 + Ckpt	216.6	0.0875
Resilient Native X10 + Ckpt + 3 Failures	233.1	0.0890

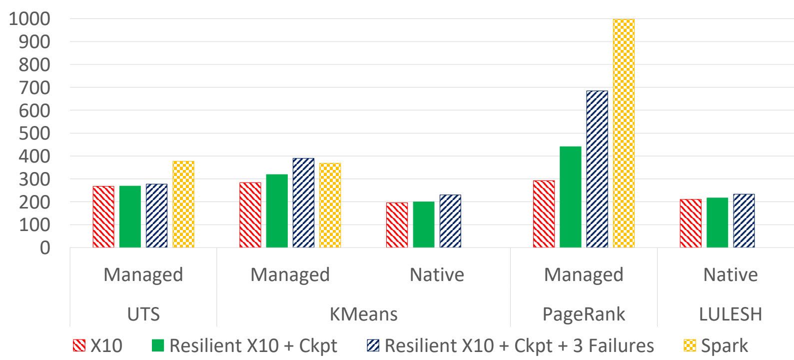

Fig. 3. Execution times for all the benchmarks (seconds)

We use a problem size of 35^3 elements per place running with 8 places.[14] At this problem size, LULESH has an average checkpoint time of 0.097 seconds. Applying Young's formula and assuming MTBF of 60 seconds yields an optimal checkpoint interval of 3.4 seconds, which corresponds to checkpointing every 38 steps. For 8 places and 35^3 elements per place, the simulation takes a total of 2,402 time steps. Resilient X10 with checkpointing takes 6.4 seconds (3%) longer than non-resilient. Of this, 6 seconds is checkpointing and 0.4 is attributable to resilient finish (0.2%). On the runs with three failures, there is an average 16.5 second (5.5 per failure) performance drop. Approximately 1.5 seconds can be attributed to failure detection and recovery of the X10 runtime system, 0.5 seconds to application-level recovery, and the remaining 3.5 seconds to lost work.

5.6 Summary

Figure 3 and Table 6 summarize the performance results across all the benchmarks and configurations. We observe that Resilient X10 always outperforms Spark. This confirms two things. First, the expressivity and level of control offered by the Resilient X10 programming model does not come at the expense of performance. Even for application kernels for which the Spark programming model is well suited, e.g., KMeans, Resilient X10 can match or exceed Spark performance. Second, Resilient X10 can deliver much higher levels of efficiency for applications that are not as well suited for Spark, e.g., UTS. In UTS, Resilient X10 has an overhead of less than 3% compared the sequential throughout, Spark is much higher at 30%. Moreover, with X10 there is the opportunity to go native, and for computationally intensive codes this is often a clear win as illustrated by the KMeans benchmark.

[14] LULESH requires a cubic number of places; to be consistent with our other experiments we run one place per node and thus have a max of 8 places possible on our 23 node cluster.

While some applications have higher resiliency overheads that others, these overheads are almost entirely due to application-level checkpointing. We measured the checkpointing costs for instance for KMeans between Resilient X10 and Spark and found them to be comparable.

Moreover, the experimental setup we have chosen—3 failures in 5-minute runs—over-emphasizes the checkpointing costs. First, the initial checkpoint is often very expensive but it is only needed once (and alternative implementations could be considered such as reloading input data from disk). With our configuration, the initial checkpoint is not amortized and amounts to a significant fraction of the execution time. Second, we implemented very frequent checkpoints to optimize for very frequent failures. With a MTBF of one day instead of one minute, the checkpointing interval (respectively overhead) would be multiplied (respectively divided) by 38. Concretely, across all four benchmarks, for a 2-hour long run with a checkpointing interval adjusted for a 24-hour MTBF, the checkpointing overhead drops below 1%. In short, in real-world use cases, we expect the resilient code to be barely slower than the non-resilient code.

Finally, we have shown that, across all the benchmarks, the downtime consecutive to a place failure never exceeds 2 seconds. In other words, 2 seconds after a failure the application code is already busy restoring data from the resilient store or even computing.

Table 6. Summary of X10 experimental results

	Managed			Native	
	UTS	KMeans	PageRank	KMeans	LULESH
X10	267.7s	283.4s	292.6s	195.9s	210.2s
Resilient X10	<268.4s[15]	291.5s	312.9s	196.1s	210.6s
Resilient Finish Overhead	<0.3%	2.9%	6.9%	0.1%	0.2%
Total Checkpointing Time	<0.7s	27.2s	127.9s	3.3s	6.0s
Runtime Recovery (1 failure)	1.5s[16]	2.0s	2.0s	1.5s	1.5s
App Data Recovery (1 failure)		13.0s	31.8s	6.5s	0.5s
Lost Work Recovery (1 failure)	1.5s	3.0s	21.0s	2.0s	3.5s
Slowdown due to Recovery (1 failure)[17]	0s	5.6s	26.3s	0s	0s
Total Recovery Time (1 failure)	3s	23.6s	81.1s	10.0s	5.5s
Resilient Application Lines of Code	627	277	319	277	4100
Changed Lines for the Iterative Framework	N/A	73	50	73	200
% Changed	N/A	26%	16%	26%	5%

6 OTHER RELATED WORK

The relationship between Resilient X10 and Big Data frameworks such as MapReduce and Spark was discussed in Section 1. Previous work on Resilient X10 was covered in Sections 2 and 4. We do not repeat those discussions here. In the following, we focus mainly on related resilience approaches for HPC applications and programming models.

[15] As the total resiliency overhead for UTS including both resilient finish and checkpointing is 0.7 seconds (0.3%), we did not separately measure the overheads of just resilient finish for UTS.

[16] Our UTS code did not separately measure the time for Runtime Recovery and App Data Recovery.

[17] The additional slowdown is mainly resulting from JVM warmup and memory management of resilient data.

HPC applications have long relied on coordinated checkpoint/restart both as a mechanism for resiliency and to decompose long-running applications into more schedulable units of work [Elnozahy et al. 2002; Sato et al. 2012]. Resilient X10 naturally supports a checkpoint/restart model by providing a resilient store abstraction and the APGAS control constructs needed to synchronize checkpoint/restart tasks across all involved `Places`. The X10 Global Matrix Library is similar to the Global View Resilience library [Chien et al. 2015] in providing globally identified distributed arrays, and in creating labelled snapshots of the data at application-controlled times for the purpose of recovery. GML does not specify special error handling interfaces as in GVR, however, capturing snapshots of the data along with X10's failure reporting support can be integrated in a flexible way for developing different failure recovery methods.

In response to increasing system scale, more loosely synchronized checkpointing approaches have been explored based on message logging and deterministic replay [Guermouche et al. 2011; Lifflander et al. 2014]. Message logging can provide a significant performance improvement over coordinated checkpointing, particularly if knowledge of ordering constraints is used to reduce the amount of information required to produce a correct replay [Lifflander et al. 2014]; furthermore, it requires little or no programmer effort to add to an application. However, it is not a flexible approach, as failures are transparent to the programmer and therefore do not allow the use of application-specific knowledge to reduce the overhead of resilience.

Approximate computing represents an alternative approach to resiliency that simply suppresses some failures based on the observation that some computations are inherently approximate or probabilistic. In some cases, analysis can be applied to obtain bounds on the distortion of discarding the results of failed tasks [Rinard 2006]. Because Resilient X10 enables the application programmer to control their fault tolerance and recovery strategies, various approximate computing approaches as well as algorithmic-based fault tolerance [Bosilca et al. 2009] can be naturally expressed in Resilient X10 as illustrated in the original Resilient X10 paper [Cunningham et al. 2014].

Designing resilient HPC programming models has been a topic of active research in recent years. MPI-ULFM (User Level Failure Mitigation) [Bland et al. 2012] is a proposal for adding fault tolerance semantics to the coming MPI-4 standard. It extends MPI-3 with failure awareness and additional interfaces for failure detection and recovery. Shrinking recovery is supported by the new interface `MPI_COMM_SHRINK` that excludes dead ranks from a given communicator. Because MPI-3 supports dynamic process creation using `MPI_COMM_SPAWN`, non-shrinking recovery mechanisms can also be implemented by spawning new ranks to replace dead ranks in a shrunken communicator [Ali et al. 2014]. Resilient X10 offers the same capabilities within the productive APGAS programming model. It uses MPI-ULFM as a low-level transport layer for scaling Resilient X10 applications to supercomputer scale as described by Hamouda et al. [2016].

Transparent recovery of APGAS applications through message logging and task replication has been recently studied in the context of the Chapel language [Panagiotopoulou and Loidl 2016]. As this work considers only side-effect-free tasks, and as the GASNet communication layer is not tolerant to process failures, further work is required to provide a complete approach to resilience in Chapel.

In the family of actor-based programming models, Erlang [Vinoski 2007] has been influential in the area of fault tolerant concurrent programming. Erlang programs benefit from user-level resilience by constructing a supervision tree between actors. A parent actor receives notifications when any of its supervisees fails, and performs the required actions for recovery. The same failure model can be expressed in Resilient X10 thanks to the nesting flexibility of the async/at/finish constructs and the provided hierarchical failure propagation through DeadPlaceExceptions. While actor placement is fixed and user-specified in Erlang, other actor-based programming models such as Charm++ [Acun et al. 2014; Kalé et al. 2011] and Orleans [Bykov et al. 2011] offer a virtual actor abstraction that hides the physical location of the actors from users and enables the runtime system to migrate the actors transparently for failure avoidance and/or load balancing. Our PlaceManager and ResilientStore abstractions apply the same virtualization concept to improve the productivity of writing Resilient X10 programs, however it does not migrate the data transparently in order to maintain the strong locality feature of the APGAS model. While Resilient X10 adopts a user-level resilience approach that enables the expression of different fault tolerance techniques at the application level, Charm++ and Orleans handle failure recovery at the runtime level transparently. Charm++ supports transparent recovery using checkpoint/restart [Zheng et al. 2012]. Orleans integrates multiple mechanisms for handling failures. It uses in-memory replication for improving the system's availability, disk-based checkpointing for restoring lost actors, and resilient transactions for handling atomic actions on multiple actors.

Partially fault-tolerant X10 implementations of lifeline-based global load balancing and Unbalanced Tree Search have been described by Fohry et al. [2015] and Fohry and Bungart [2016]. Whereas those implementations can fail due to loss of a single place if the failure hits at the worst possible time, our implementation described in Section 5.2 is resilient to any failure of a single place except for place zero.

7 CONCLUSIONS

This paper describes the evolution of Resilient X10 into a powerful and practical programming framework for implementing high performance distributed and resilient applications. While the Resilient X10 semantics remain the foundation of this work, the lack of data resilience in the original programming model design drastically limited its usefulness. Conversely in-memory data grids such as Hazelcast lack a rich tasking model capable of orchestrating parallel and distributed computations. In this work, we combine the two in a seamless way: the data and control semantics obey the happens-before invariance principle; heap and resilient stores are organized according to the same PGAS abstraction.

New capabilities such as elasticity and fully integrated standard library support for non-shrinking recovery provide powerful new options to the application programmer. These capabilities significantly reduce the complexity of implementing stateful applications designed to survive failure and preserve the core productivity and performance benefits of the APGAS programming model.

As further developed in this paper, the Resilient X10 programming model naturally supports Big Data paradigms such as those supported by MapReduce or Spark. In addition, Resilient X10 also supports classes of applications with complex distributed communication patterns, shared mutable distributed state, and dynamic fine-grained work generation. The Resilient X10 model also enables a spectrum of recovery techniques ranging from checkpoint/restart, to resilient data structures, to approximate computing and algorithmic fault tolerance. We strongly believe this generality and flexibility is essential to accelerate the adoption of datacenter-scale computing infrastructure in an ever-increasing number of application domains.

ACKNOWLEDGMENTS

The Resilient X10 research was funded in part by the U. S. Air Force Office of Scientific Research under Contract No. FA8750-13-C-0052. Work on the LULESH application was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923. Source code line counts were generated using David A. Wheeler's 'SLOCCount'.

REFERENCES

Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kalé. 2014. Parallel programming with migratable objects: Charm++ in practice. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*. IEEE, 647–658.

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant stream processing at Internet scale. *Proc. VLDB Endowment* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

Md Mohsin Ali, James Southern, Peter Strazdins, and Brendan Harding. 2014. Application level fault recovery: Using fault-tolerant Open MPI in a PDE solver. In *2014 International Parallel & Distributed Processing Symposium Workshops*. IEEE, 1169–1178.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: relational data processing in Spark. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1383–1394. https://doi.org/10.1145/2723372.2742797

Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. 2012. An evaluation of user-level failure mitigation support in MPI. In *Proc. Recent Advances in Message Passing Interface – 19th European MPI Users' Group Meeting (EuroMPI '12)*. Springer, 193–203.

George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. 2009. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.* 69, 4 (April 2009), 410–416. https://doi.org/10.1016/j.jpdc.2008.12.002

Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endowment* 3, 1-2 (2010), 285–296.

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proc. 2nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The new adventures of old X10. In *Proc. 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. 51–61. https://doi.org/10.1145/2093157.2093165

Chapel 2016. *Chapel Language Specification version 0.982*. Technical Report. Cray Inc.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '05)*. 519–538. https://doi.org/10.1145/1094811.1094852

Andrew Chien, Pavan Balaji, Peter Beckman, Nan Dun, Aiman Fang, Hajime Fujita, Kamil Iskra, Zachary Rubenstein, Ziming Zheng, Rob Schreiber, et al. 2015. Versioned distributed arrays for resilience in scientific applications: Global View Resilience. *Procedia Computer Science* 51 (2015), 29–38.

Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Killian. 2013. EventWave: programming model and runtime support for tightly-coupled elastic cloud applications. In *Proc. 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2523616.2523617

Silvia Crafa, David Cunningham, Vijay Saraswat, Avraham Shinnar, and Olivier Tardieu. 2014. Semantics of (Resilient) X10. In *Proc. 28th European Conference on Object-Oriented Programming*. 670–696. https://doi.org/10.1007/978-3-662-44202-9_27

David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. 2014. Resilient X10: efficient failure-aware programming. In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice Of Parallel Programming (PPoPP 2014)*. ACM, 67–80. https://doi.org/10.1145/2555243.2555248

Doug Cutting and Eric Baldeschwieler. 2007. Meet Hadoop. In *O'Reilly Open Software Convention*. Portland, OR.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proc. 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*. 10–10.

E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey* 34, 3 (2002), 375–408.

Claudia Fohry and Marco Bungart. 2016. A robust fault tolerance scheme for lifeline-based taskpools. In *45th International Conference on Parallel Processing Workshops (ICPPW)*. 200–209. https://doi.org/10.1109/ICPPW.2016.40

Claudia Fohry, Marco Bungart, and Jonas Posner. 2015. Towards an efficient fault-tolerance scheme for GLB. In *Proc. ACM SIGPLAN Workshop on X10 (X10 '15)*. ACM, New York, NY, USA, 27–32. https://doi.org/10.1145/2771774.2771779

Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. 2011. Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 989–1000.

Sara S. Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, and Olivier Tardieu. 2016. Resilient X10 over MPI User Level Failure Mitigation. In *Proc. ACM SIGPLAN Workshop on X10 (X10 '16)*. https://doi.org/10.1145/2931028.2931030

Sara S. Hamouda, Josh Milthorpe, Peter E Strazdins, and Vijay Saraswat. 2015. A resilient framework for iterative linear algebra applications in X10. In *Proc. 16th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2015)*. https://doi.org/10.1109/IPDPSW.2015.14

Hazelcast, Inc. 2014. Hazelcast 3.4. https://hazelcast.com/

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. 2010 USENIX Annual Technical Conference*. 11–11.

Maja Kabiljo, Dionysis Logothetis, Sergey Edunov, and Avery Ching. 2016. *A comparison of state-of-the-art graph processing systems*. Technical Report. Facebook. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/

Laxmikant V. Kalé, Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataraman, Lukasz Wesolowski, and Gengbin Zheng. 2011. *Charm++ for Productivity and Performance: A Submission to the 2011 HPC class II challenge*. Technical Report. Parallel Programming Laboratory.

Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973.

Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A compiler-free PGAS library. In *Proc. 8th International Conference on Partitioned Global Address Space Programming Models*. Article 5.

Jonathan Lifflander, Esteban Meneses, Harshitha Menon, Phil Miller, Sriram Krishnamoorthy, and Laxmikant V Kalé. 2014. Scalable replay with partial-order dependencies for message-logging fault tolerance. In *Proc. IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Madrid, Spain, 19–28.

Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theor.* 28, 2 (March 1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the Cloud. *Proc. VLDB Endowment* 5, 8 (April 2012), 716–727.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. 135–146. https://doi.org/10.1145/1807167.1807184

Josh Milthorpe, David Grove, Benjamin Herta, and Olivier Tardieu. 2015. *Exploring the APGAS programming model using the LULESH proxy application*. Technical Report RC25555. IBM Research.

Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: an unbalanced tree search benchmark. In *Proc. 19th International Conference on Languages and Compilers for Parallel Computing (LCPC'06)*. Springer-Verlag, Berlin, Heidelberg, 235–250.

Konstantina Panagiotopoulou and Hans-Wolfgang Loidl. 2016. Transparently resilient task parallelism for Chapel. In *2016 International Parallel & Distributed Processing Symposium Workshops*. IEEE, 1586–1595.

John T. Richards, Jonathan Brezin, Calvin B. Swart, and Christine A. Halverson. 2014. A decade of progress in parallel programming productivity. *Commun. ACM* 57, 11 (Oct. 2014), 60–66. https://doi.org/10.1145/2669484

Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proc. 20th Annual International Conference on Supercomputing (ICS '06)*. 324–334.

Vijay Saraswat, Gheorghe Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. 2010. The Asynchronous Partitioned Global Address Space Model. In *Proc. First Workshop on Advances in Message Passing (AMP'10)*.

Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-based global load balancing. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. 201–212. https://doi.org/10.1145/1941553.1941582

Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. 2012. Design and modeling of a non-blocking checkpointing system. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis 2012 (SC '12)*.

Richard D. Schlichting and Fred B. Schneider. 1983. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.* 1, 3 (Aug. 1983), 222–238. https://doi.org/10.1145/357369.357371

Avraham Shinnar, David Cunningham, Benjamin Herta, and Vijay Saraswat. 2012. M3R: Increased performance for in-memory Hadoop jobs. In *Proc. VLDB Endowment (VLDB '12)*.

Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. 2014. X10 and APGAS at Petascale. In *Proc. 19th ACM SIGPLAN Symposium on*

Principles and Practice Of Parallel Programming (PPoPP 2014). ACM, 53–66.

The X10 Language 2019. Git Repository. git@github.com:x10-lang/x10.git

Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. 4th Annual Symposium on Cloud Computing (SOCC '13)*. Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

Steve Vinoski. 2007. Reliability with Erlang. *Internet Computing, IEEE* 11, 6 (2007), 79–81.

Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

X10 Applications 2019. Git Repository. git@github.com:x10-lang/x10-applications.git

X10 Benchmarks 2019. Git Repository. git@github.com:x10-lang/x10-benchmarks.git

X10 v2.6.1 2017. X10 2.6.1 Release. https://doi.org/10.5281/zenodo.822471

Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2014. GraphX: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394* (2014).

Chaoran Yang, Karthik Murthy, and John Mellor-Crummey. 2013. Managing asynchronous operations in Coarray Fortran 2.0. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1321–1332.

John W Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (1974), 530–531.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 15–28.

Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. 2014. GLB: Lifeline-based global load balancing library in X10. In *Proc. First Workshop on Parallel Programming for Analytics Applications (PPAA '14)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2567634.2567639

Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. 2012. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Proc. IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 1–6.

Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS extension for C++. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1105–1114.