Parallel Computing 40 (2014) 136–156

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

X10-FT: Transparent fault tolerance for APGAS language and runtime

Zhijun Hao [a], Chenning Xie [b], Haibo Chen [b,*], Binyu Zang [b]

[a] School of Computer Science, Fudan University, China
[b] Institute of Parallel and Distributed Systems, School of Software, Shanghai Jiao Tong University, China

ARTICLE INFO

Article history:
Available online 7 December 2013

Keywords:
Asynchronous partitioned global address space (APGAS)
Fault tolerance
X10

ABSTRACT

The asynchronous partitioned global address space (APGAS) model is a programming model aiming at unifying programming on multicore and clusters, with good productivity. However, it currently lacks support for fault tolerance (FT) such that a single transient failure may render hours to months of computation useless.

In this paper, we thoroughly analyze the feasibility of providing fault tolerance for APGAS model and make the first attempt to add fault tolerance support to an APGAS language called X10. Based on the analysis, we design and implement a fault-tolerance framework called X10-FT that leverages renowned techniques in distributed systems like distributed file systems and Paxos, as well as specific solutions based on the characteristics of the APGAS model to make checkpoints and consensus. This allows the system to transparently handle machine failures at different granularities. Using the features of the APGAS model, we extend the X10 compiler to automatically locate execution points to checkpoint program states without any intervention from programmers. Evaluation using a set of benchmarks shows that the cost for fault tolerance is modest.

1. Introduction

The emergence of multicore machines has made exploiting parallelism a necessity to harness the abundant computing resources in both a single machine and clusters. This, however, may hinder programming productivity as multi-threaded and distributed programming are hard to use correctly and concurrency/distributed bugs are hard to spot.

The asynchronous partitioned global address space (APGAS) model [1] attempts to ease programming on both cluster and multicore machines. This model is an extension of PGAS [2]. PGAS abstracts a platform as a global yet partitioned address space, where each entity (e.g., core or machine) has its own portion of address space, yet can directly access other portions of address space using special language constructs. APGAS extends the PGAS model with "asynchrony", supports heterogeneous hardware and intends to provide programmers with higher productivity. Specifically, there are two concepts in APGAS, the *Place* and the *Async*, to allow dynamically spawning tasks and bridge machine heterogeneity. APGAS has a richer execution framework than the SPMD style generally used by PGAS, because of the underlying heterogeneous hardware. Generally, the program pattern is single-threaded in each task in PGAS model. They also use barriers to synchronize all the tasks. The APGAS model allows each node to execute multiple tasks from a task pool, and allows nodes to invoke work on other nodes. The APGAS model also provides richer language constructs and more powerful compiler and runtime than PGAS to express the asynchrony and to improve productivity. Using these constructs, good structured parallelism can be achieved

* Corresponding author.
E-mail addresses: haozhijun@fudan.edu.cn (Z. Hao), xiechenny@sjtu.edu.cn (C. Xie), haibochen@sjtu.edu.cn (H. Chen), byzang@sjtu.edu.cn (B. Zang).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.11.006

in more simple, clear and structured user code. A recent embodiment of the APGAS model is the X10 language [3], which hides underlying machine heterogeneity from users and allows users to conveniently write multi-threaded programs that can be executed in cluster environments. A number of other programming models, including MapReduce [4], can be easily expressed in X10 [5].

Unfortunately, though the APGAS model has the potential of embracing both performance and productivity, there is currently no support for fault tolerance in known languages and runtime. Providing fault tolerance (FT) is important as many current computation tasks (like those in HPC) usually require running several days or even months on thousands of cores. Hence, even a small failure in a small component may render the whole computation task meaningless, and it requires a restart of this failed computation task or even all tasks. With the increasing scale of machines, the potential error rate grows as well [6], which makes the problem even more serious.

Although there is some previous work in providing FT in PGAS model, such work cannot be easily used in X10. There are two different policies to provide FT support in PGAS, one is using computation redundancy, and the other is using storage replication, such as the data checkpoints. In X10-FT, we use the classic checkpoint-recovery method to provide FT support. Compared with PGAS, the FT in APGAS has both new challenges and opportunities. The main challenge is to handle the complicated state when building the checkpoints. As the PGAS programs are usually SPMD style, which means that usually the programs are single-threaded, and there are some barriers to synchronize all these processors. Hence in PGAS, the state that needs to be recorded is simple and could be easily get in a consistent way at barriers. APGAS provides more asynchrony, which means that there may be multiple asynchronous tasks executing in each node and one node can invoke work on other nodes. The points at which checkpoints are consistent are hard to find. We extend the X10 runtime to solve this hard problem by doing runtime checks. Meanwhile, because APGAS aims to provide high productivity, richer language features can be used. The X10 compiler and runtime are also more powerful. This leads to clear and simple user code with structured parallelism, which could make the checkpoint code transparent to programmers, as some analyses can be done by the compiler automatically.

In this paper, we make the first comprehensive analysis on providing fault tolerance to an APGAS-based language and runtime using checkpoint-recovery method, by using X10 as an example. The goal is to see how renowned techniques in distributed systems and APGAS-specific features may help to provide reliable and efficient fault-tolerance computation in the APGAS model.

As a typical APGAS language, X10 has a limited set of features related to fault tolerance. First, in each place, X10 maintains an exception system similar to Java, any exception in the user code will be caught by X10 runtime. Hence, we can mainly focus on making the X10 runtime fault tolerant, while leaving the faults in user code to be handled by X10 runtime. Second, the APGAS model ensures that a global variable should only be accessed in its own home place, who possesses the global variable during the whole execution. Local data among tasks are accessed through different copies with the help of X10 runtime. Hence, it is possible to selectively redo a task by eliminating side effects on the global variables of the system in that task. Finally, there are a set of explicit synchronization primitives, such as *finish*, *collecting-finish* and *at (P)*. These primitives help switch the execution flow between user code and runtime code, so that fault detection and recovery can be mainly implemented in the X10 runtime with little or no modification to user code.

X10-FT leverages the language features of APGAS and combines them with known techniques in distributed system for fault tolerance. Based on the key primitives such as *finish*, we modify the X10 compiler to automatically insert checkpoint code into user code. X10-FT also adds an analysis pass in the X10 compiler to identify necessary variables that need to be recorded in each checkpoint. Besides, we still provide the flexibility such that users can also mark precise variables that need to be recorded manually using compiler annotations. To reliably store checkpoints for recovery, X10-FT seamlessly incorporates a distributed file system (DFS) engine into X10 runtime. X10-FT leverages the Paxos [7] consensus protocol to reliably detect possible node failures or network partitions. When failures are detected, X10-FT resumes the disrupted activities by rebuilding the failed place on another node, recovering the place state from the latest valid checkpoints in DFS, and changing the control flow of these activities to that in the checkpoint to continue their execution.

Currently, we have implemented a working prototype of X10-FT, including extensions to X10 compiler and runtime, incorporation of the Hadoop distributed file systems (HDFS) [8] for storing checkpoints and ZooKeeper [9] for consensus of failures. This prototype can run real-world X10 programs with fault-tolerance support. To evaluate the overhead of providing fault tolerance in X10, we use WordCount [4], a typical map-reduce application in distributed system, SSCA#1 [10], an application for bioinformatics optimal pattern matching that stresses integer and character operations, and two benchmarks from the HPC challenge benchmark suite [11], the Global RandomAccess and STREAM. Our evaluation shows that X10-FT can successfully recover the tested programs with only modest performance overhead.

This paper makes the following contributions:

- The first framework that provides APGAS programs with efficient fault-tolerance support.
- Combine renowned techniques in distributed system, like Paxos and DFS, with APGAS features in X10 runtime and compiler to find points to do consistent checkpoints and provide fault tolerance, which is transparent to programmers.
- A detailed implementation of X10-FT.
- A detailed evaluation and analysis of overhead using X10-FT with different kinds of benchmarks.

The rest of this paper is organized as follows. First, we discuss the related work of X10-FT, especially those using the PGAS model in Section 2. Section 3 introduces the background of X10 and fault tolerance. Sections 4 and 5 present the design and

implementation of X10-FT. Section 6 evaluates the effectiveness and overhead of fault tolerance using X10-FT. Finally, we conclude this paper with a brief remark on our future work.

2. Related work

A lot of fault tolerance techniques have been created to make applications in HPC domain recover from frequent failures efficiently. The most popular rollback-recovery approach uses checkpoint/restart in HPC environments [12]. Other techniques include, the algorithm-based fault tolerance (ABFT) approach [13,14], the proactive fault tolerance approach [15,16], and so on. The checkpoints used for recovery can be diskless [17,18], and also can be at the user/kernel level. The checkpoint/recovery code can be inserted either by the programmer manually [19], or by the compiler automatically [20], or by using the existing software infrastructure, such as BLCR (Berkeley Labs Checkpoint Restart) [21]. X10-FT leverages the classical checkpoint/restart approach to make X10 programs fault tolerant. These checkpoints are at application level, instrumented by the X10-FT compiler automatically, which makes them transparent to X10 programmers, and are saved into the underlying DFS in clusters.

This paper is the first attempt to utilize the features of APGAS model to make X10 programs fault resilient. There is no fault tolerance in APGAS model at present, and most of the related research focuses on the PGAS model and MPI.

There are a few efforts to provide fault tolerance to programs of the PGAS model. Ali et al. [22] leverage shadow data structures and redundant remote memory accesses to make PGAS programs tolerate potential faults, and their implementation uses Global Arrays language [23,24]. In contrast, X10-FT uses checkpoints.

In [25], Vishnu et al. provide a fault-resilient, one-sided communication runtime framework, FT-ARMCI, for PGAS model. It also uses the Global Arrays language and its communication runtime, ARMCI [26]. The FT-ARMCI provides a fault detection module by leveraging the RDMA technique, a fault-resilient process manager and other collective communication primitives to tolerate faults. In contrast, the runtime of X10-FT uses heart-beat messages to detect faults, and is the first one that using the Paxos protocol and DFS in the PGAS to make the system more robust.

There are also systems using checkpoints to provide PGAS fault tolerance. Scarpazza et al. [27] use system-level virtualization software, such as Xen, and high-speed network, such as the InfiniBand, to provide fault tolerance. The general idea is to automatically identify the global recovery lines, at which the system reaches a quiescent, global consistent state, and do checkpoints at that time point. These global recovery lines are easy to find due to the SPMD program style of PGAS. During failures, the checkpoints, which may be an entire OS image including the application, could be migrated to another node using facilities of the virtualization software. It relies heavily on many functions of the virtualization software and the capability of the underlying network, and is more heavyweight. The virtualization software and high-speed network are also not available in every cluster. X10-FT does not have such requirements.

Tipparaju et al. [28] propose to provide fault tolerance to Global Arrays programs using user-coordinated checkpoints by recording only the global data. In contrast, X10-FT provides transparent fault tolerance to programmers and can handle the local data. This is due to the benefits of richer language features and simple and structured user code in APGAS. Meanwhile, the powerful X10 compiler and runtime help a lot. The points to do consistent checkpoints can be easily found in PGAS model because of the SPMD program style, while in APGAS, because of its asynchrony feature, these points are not so easy to find. Further, X10-FT integrates techniques in distributed systems like Paxos and DFS for more reliable failure detection and recovery. Dinan et al. [29] propose to do selective recovery from full-state checkpoints to recover only the failed tasks. This method can also be used in X10-FT.

There are also a number of efforts on providing fault tolerance for MPI programs, such as CoCheck [30], FTC-Charm++ [31] and FT-MPI [32]. However, the fault tolerance is implemented at the low API level. The MPI programming model can be easily expressed in X10, and the semantics of X10 language is more powerful than standard MPI, especially for the global address space and activity models. Hence, X10-FT directly provides fault tolerance support based on X10 semantics, which essentially provides fault tolerance to MPI programs written in X10 as well.

Further, virtualization has also been used in HPC [33,34] to provide fault tolerance. This, however, is a more intrusive solution that requires adding virtualization layer to the running systems, either constantly or on-demand.

3. Background

This section reviews some of the features of X10 runtime and the key X10 language constructs, which are related to X10-FT. We use the WordCount [4] program as an example. This section also briefly discusses fault-tolerance techniques in distributed systems.

3.1. X10 runtime and language constructs

The two key concepts in APGAS are *Place* and *Activity*. In X10, a place is an independent entity, which consists of two processes: the launcher and the runtime. While an activity is a task that contains user code. The activities in an X10 program will be distributed to worker threads in all places, which partitions the entire global address space in a cluster.

Inside each place, the launcher process is in charge of affairs management of that place, such as starting or exiting that place, forwarding the outputs in the runtime process to the user, helping its runtime build links (e.g., sockets) with other runtimes, and thus other places, and so on. Launchers are built according to the architecture shown in Fig. 1, which is a full binary tree. A parent launcher in the tree is responsible for forking the launcher children, either on its own node or on other nodes. The communications between a parent and its children are through a socket link and two pipes. The socket is used to transfer control messages, while the two pipes are used to transfer the "stdout" and the "stderr". The control messages include the ones that help build links between each pair of runtime during initialization, and the ones that control the termination of child places. The "stdout" output and the "stderr" output in each place are forwarded through the tree architecture layer by layer and finally merged in the "stdout" and "stderr" of the starting process, respectively. The runtime process in each place provides the user code with the execution engine, which is responsible for initializing the execution environment, scheduling tasks, and so on.

Fig. 2 illustrates the architecture of X10 programs. Each place has a worker-thread pool consisting of several workers. The communications between places are made through message-passing. In the APGAS model, activities are allowed to be dynamically spawned. An activity will get executed by a worker in the destination place synchronously or asynchronously. An activity is called a *remote activity*, if it is pushed directly into another place after spawned in a place. For instance, the "main" function in the user code will be pushed into the main place - place 0, as an activity. Then the "main" activity will assign tasks to other places by pushing remote activities into them. The activities spawned by a remote activity, either executed in the same place or pushed into another place, can be considered as its child activities. In the APGAS memory model, the global data are partitioned into different parts in each place, as shown in Fig. 2. A place can only access the global data on another place by sending remote activities to it. The remote activity helps to read or write these data, and will send the results back, if necessary. The local variables inside a remote activity are only visible in that activity.

In the following, we use WordCount, a typical data-parallel application, as the example to illustrate the language constructs in X10. Fig. 3 shows the pseudo code of WordCount written in X10:

(a) The ***async*** {S} statement is an important language construct in X10, which launches a local activity that consists of statement S in the current place, without blocking the current activity executing this *async* statement.
(b) The ***finish*** {S} statement is a barrier at which the current activity will wait for all activities spawned inside statement S to terminate.
(c) The ***at (p)*** {S} statement appoints a specific place "p" to execute the remote activity consisting of statement S.
(d) The *Collecting-finish* statement, whose syntax is ***finish (r)*** {offer (Intermediate result T)}, is a framework in X10. It is similar with the MapReduce [4] model. This framework is a typical application of the above language constructs. During execution, it collects the intermediate results in each place, and then automatically reduces them into a final result at the initiator place by executing a user-defined *Reducible* function.

3.2. Fault tolerance in distributed systems

The requirements of communication and consistency in the presence of the geographical distances make a distributed system complex. Fault tolerance in a distributed system is even harder, where replication is a general solution to achieve fault tolerance. Generally, there are two categories of replication-based fault tolerance in a distributed system: *computation redundancy* and *storage replication*. The former may have multiple identical computation tasks executed simultaneously, and an arbitrator may just choose the fastest one, or get a consistent result from all of them, as the systems in [35,36] did. It may lead to performance loss, or require additional computing resources. Storage replication instead replicates data into many

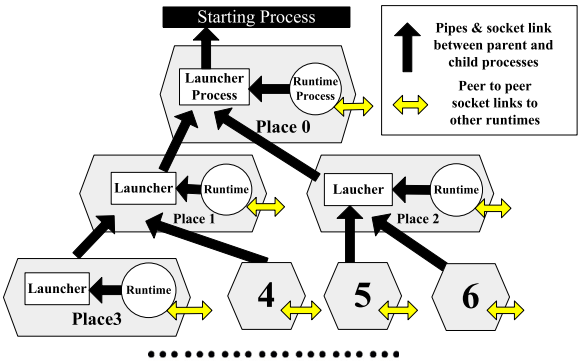

Fig. 1. X10 place architecture.

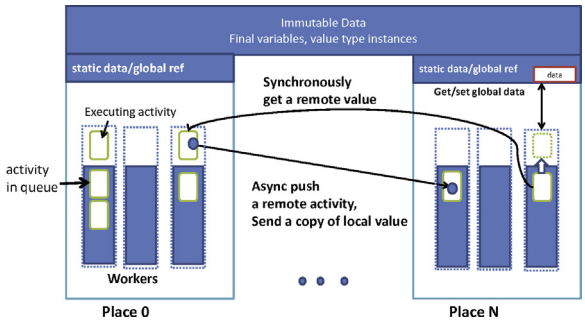

Fig. 2. Architecture of X10 programs.

```
file_data = stat(word_files);
total_result =
finish(Reducer r) {
  for (p in Places) {
    place_input =
      split(file_data);
    async at(p) {...}
  }
}
print(total_result);
```

```
async at(p) {
  finish for (i in Actives) {
    activity_input =
      split(place_input);
    async activity_res =
        map(activity_input);
  } //checkpoint
  place_res = reducer(activity_res);
  offer(place_result);
} //checkpoint
```

```
reducer(activity_res) {
  while() {
    finish {
    for(i in half_activities)
    async  reduce(
      2i_activity_result,
      2i+1_activity_result);
    } //checkpoint
  }
}
```

Fig. 3. WordCount pseudo code.

copies on different storages to ensure durability of data and/or checkpoints. There are a lot of systems using storage replication to tolerate faults, such as fail-safe PVM [37], CoCheck [30] and Starfish [38].

Most of existing distributed programming models have their own fault tolerance models. For instance, the Hadoop Map-Reduce system just restarts the failed work if that work encounters a fault. This is feasible because in Map-Reduce model, the tasks are loosely coupled, which means that each task is an independent job and does not affect others. However, the APGAS model is more powerful and allows user to construct more complicated programs. Hence, adding fault tolerance support to the APGAS model is more challenging due to its abundant parallelism.

4. Design

X10-FT adds fault tolerance support to the APGAS model in several aspects. First, it improves the detection capability of view changes (failed places) in the original X10. The detectors should be distributed themselves, in order to survive single point of failures. Each place connects with one of the detectors using heartbeat messages, which are able to detect failures in time. X10-FT incorporates the Paxos [7] protocol to make consensus of the current view of places among detectors when there are split-brain syndromes due to a network partition, or when there are some places crashed. Second, X10-FT uses checkpoints to store computation states of tasks and place status of an X10 place into the underlying distributed file system (DFS) for recovery. Besides re-executing all un-executed tasks of the failed place, X10-FT also recovers unfinished computation by taking checkpoints of running tasks at proper time. These checkpoints could be automatically set according to the key X10 language constructs, such as the *finish{}* and *at (p){}*, or manually set by user annotations. X10-FT also provides a library to interact with the underlying distributed storage. All these features are transparent to users.

The entire procedure is as follows. At each proper point, X10-FT will capture states of a place and the running tasks in that place automatically, then save them into the underlying distributed storage. When there are hardware failures or some other faults that crash the places on a node, the Paxos protocol with heartbeat messages helps detect these events in time. Then the parent place of the crashed one is notified and rebuilds the crashed place on an available machine. The rebuilt place will load saved states into memory from the checkpoint records in the distributed storage, and resume the execution of tasks as captured in the checkpoint.

4.1. Failure model

X10-FT is designed to tolerate fail-stop failures of arbitrary number of places (except the place 0), where the failed process stops working and all data associated with the failed process are lost. It does not handle non-fail-stop failures, such as the byzantine faults or the inherent defects of a program. Specifically, any failures that will make a place crash, such as hardware

failures, and network partition, can be handled by X10-FT effectively. The network partition problem is handled by X10-FT with the help of the Paxos protocol. X10-FT treats the smaller group of disconnected machines that is the minority as the failed ones that have crashed. As for other failures, such as the bit flips, X10-FT cannot handle them directly unless they make the place crash. If these failures cause exceptions, they can be handled by the exception-handling system of X10 if the programmer has set proper exception handlers. Further, X10-FT mainly focuses on *output-deterministic* applications, such as the scientific computing programs. The word "output-deterministic" is borrowed from the paper [39], which means that no matter how many times the application is re-executed, it would get the same final results if the program input is identical despite the different execution orders of tasks. A lot of X10 programs are output-deterministic, because now the X10 language is mainly used in the HPC domain.

In the current design, based on the binary tree architecture of places, the parent place of a failed place is responsible for handling the failures, including rebuilding a new place and recovering the states of the crashed one in the new one. As shown in Fig. 1, the place 0 is the root of the tree architecture, which is usually in charge of managing all the places' executions. At this time, failures of the place 0 are unrecoverable. This is because it has no parent in the place architecture (see Fig. 1), which would be responsible for recovering it after failures. Also, because place 0 is the execution manager, the states and control flows inside place 0 are more complicated than those in other places, are more difficult to recover. If there is a network partition between place 0 and others, there is also no way to make another place become the execution manager at present. Further, because place 0 usually runs on the local machine where the user launches the program, if place 0 crashed, it means that machine may have some problems, and the user could know this in time. So taking the above points into account, even though we could take more efforts to recover place 0, we choose to make the entire program crash if place 0 has a fault at present. X10-FT can tolerate otherwise failures of multiple places. If a few places crashed simultaneously, the parent of each will rebuild them respectively.

We currently do not buffer I/O requests between checkpoints. As we have observed that, usually in X10 programs, the I/O requests are isolated inside each place, especially for storage and display devices. For example, usually each place only operates on its own persistent storage. Since each task is output-deterministic, the re-execution of I/O requests to storage will just rewrite the same result once again. For network requests, because each message is assigned a unique ID when created, the receiver will find that a message is a duplicated one if it has already been received. Then this duplicate message will be handled according to previous task-execution records in the receiver place. There may be also some I/O operations for display devices. Hence, if a task is rolled back to the previous checkpoint and continues its execution from that point, users may observe the same output printed again as that has been done before the place crashed. However, the final output will still be consistent. In some rare situations, if the I/O requests affect some global states shared between places, it indeed may lead to a wrong result. So in such cases, X10-FT needs to consider the I/O requests as part of global states, and use some techniques to make the system states consistent before and after failures, such as buffering the requests or using version numbers. At present, we just implement the simple version.

4.2. Checkpoint

X10-FT includes an enhanced X10 compiler and runtime. The compiler does a liveness analysis to automatically find out live variables at checkpoints. These checkpoints are spotted by the compiler according to key X10 language constructs. Then at run time, the enhanced X10 runtime will check whether this point is consistent for doing checkpoints. The live variables include the global ones in a place, and the local ones in a remote activity. The compiler also automatically inserts a checkpointing code snippet into the user-written X10 code at these proper positions. Meanwhile, the compiler will insert code snippet into the user code to record the control flow of the program at run time.

In addition to the above states, X10-FT also records messages among places at checkpoints. These messages are handled in a distributed way, that is, each place manages its messages independently. In each place, the X10-FT runtime records every new message in the memory. More precisely, it records all sent messages, and the unique ID of each received message. There are two cases when it is useful. First, if the destination crashes, the origin will be able to re-send a previously recorded message. Second, when the origin recovers after a failure, the destination will be able to tell the difference between a new message and a duplicated one from the origin. We observed that in X10 applications the order of creation of tasks in each place is deterministic, and we assign a unique ID for each task when it is created. According to the type and contents of a received message, the X10 runtime will take corresponding actions. Such actions include launching a remote activity (task), which is actually a closure, to register it for execution in the specified place. The status of a task received in a place, for example, whether it has been executed in that place, or whether it has finished the execution in that place, and the status of a task sent out in a place, for example, whether its reply message has been returned back from the destination place, are also kept in the corresponding message queue. The total number of each kind of messages, namely, transmitted messages (tasks sent to others), the received messages (tasks received from others), and the reply messages (replies of a received task), is counted. The relations between the received messages and the reply messages are also maintained.

When a remote activity finishes its execution in a place, it checks the status of the task pool of that place and checkpoint the status of that place, including the message queues and the global variables, if at that time the task pool is empty. Because this implies that at that time, the status of the place is *consistent*. Launching a remote activity in place *p* corresponds to the *at (p){}* language construct in X10 language. When doing checkpoints at the points where a remote activity has finished its execution, the local variables in that remote activity need not to be saved, as they are only visible inside that remote activity.

During the execution of a remote activity, checkpoints are inserted at the sequential parts of the execution, for example, the points where the *root finish{}* language constructs, namely, the top level *finish{}* construct, in an *at (p)* construct finishes its execution. These points are actually local barriers in a remote activity. According to our experience, a remote activity with multiple *finish* constructs usually runs for a long time, and is worth doing internal checkpoints. It also needs to check the status of the task pool when doing the internal checkpoints inside a remote activity to make sure these checkpoints are *consistent*. When doing internal checkpoints, all the live local variables at that time should also be saved in the checkpoints.

The above kinds of checkpoints are identified automatically by the compiler according to X10 statements in the source code, such as *at (p)*, *finish* and *collecting-finish*. There are also some checkpoints inserted by the X10-FT runtime at run time when other kinds of messages received finish their executions. Each checkpoint record saved is uniquely indexed by the associated task ID and an increasing checkpoint number. At run time, when an X10 program of the fault-tolerant version that is generated by the X10-FT compiler reaches these checkpoints, the above-mentioned data will be flushed into the underlying DFS from memory to make the checkpoint records persistent. Generally, different tasks in the same place are executed simultaneously. They could affect each other when accessing the same global variable managed in that place. The consistent checkpoint records make sure that the global variables at that time are also consistent. Meanwhile, in each place, the checkpoint records of the message queues and the global data are incremental, and it significantly reduces the space overhead. At present, the *finish* checkpoint records in each task are not incremental, which will be our future work. The *root finish{}* statement, or *collecting-finish* statement, in place 0 usually acts as a global barrier among all the places. It could be used to reduce the overall checkpointing overhead by doing a global consistent checkpoint in each place simultaneously at that time and then cutting previous checkpoints out.

4.3. Recovery

The X10-FT framework will enter recovery mode when the detector reports a failure when a node crashes or a network partition happens. When a place crashes, a change of membership (view) will be detected by the heartbeat mechanism. Through the Paxos protocol, the remaining available places can get a new consistent view of the alive places. These places then close the connections between them and the failed one. The parent place of the failed place in the tree architecture (see Fig. 1) is responsible for rebuilding that crashed place on an available machine. After the failed place has been rebuilt successfully, that new place will send heartbeat messages to its detector and this change of view, that is, adding a new member, will also be broadcasted to all the places through the Paxos protocol. Then the new place will recover the connections between itself and others.

The rebuilt place will first load the most recent checkpoint record from the underlying DFS, then it will check the consistency of this record to see whether it is valid. In the worst case, if the latest record in a place is not valid and there is no other *consistent* records, this place has to be restarted from scratch. After the states of the rebuilt place are restored to those in the chosen checkpoint record, it will send requests to other places to fetch the tasks lost since that checkpoint. These tasks include the unfinished ones at the checkpoint, and the new ones which have not been received. Unfinished tasks without a saved checkpoint record are simply re-executed. Unfinished tasks with a valid checkpoint will continue its execution from that checkpoint, because X10-FT has recorded its execution flow in the record. Finished tasks are skipped. New tasks are simply executed. During recovery, in order to make the execution order of tasks received consistent with the semantics of the application, that is, meeting the requirements of the local barriers in a place and the global barriers among all places, the enhanced runtime will execute these tasks according to their associated ‘‘sentinels’’.

Based on our assumption, namely, the application is output-deterministic, the recovered task will send duplicate messages to other places, because its execution is restarted from the checkpoint. In addition, these messages may be sent in a different order compared with that in the previous execution. However, there is no problem for this case in X10-FT, as other places will find these messages have already been received by searching their received message queues. If a task has already finished its execution in a place, it won't be re-executed in that place again. The reply message associated with this task, which is saved in the reply message queue in that place, if there is any, will be sent back to the rebuilt place. If a task has not been received yet in a place due to network problems, it is treated as a new one in that place. If a task has been received but has not got executed yet, or has not finished its execution, this duplicate message is simply discarded.

5. Implementation

Currently, we have implemented a working prototype of X10-FT based on X10 version 2.2.1. This prototype incorporates ZooKeeper [9] to detect failures; integrates HDFS [8] under X10 layer to reliably save the checkpoint records for recovery. We have added place-rebuilding support to the X10 runtime. We have also enhanced both the X10 compiler and X10 runtime to make X10 programs support recording and recovering the states, namely, the messages, the intermediate data, and the control flow.

5.1. Incorporating HDFS

The DFS engine chosen in X10-FT is the Hadoop distributed file system (HDFS), which is a mature open-source DFS that has been widely used in many fields. A set of interfaces through which X10 programs can interact with the DFS have been

added into X10 source code. We use annotations provided in the X10 compiler to make X10 code utilize C++ code in the HDFS library.

5.2. Integrating ZooKeeper

ZooKeeper is an open source tool that is widely used to monitor status of nodes in distributed systems. ZooKeeper provides a heartbeat mechanism to detect failures, allows data sharing among the replicas, and integrates the Paxos protocol to make consensus. These features make it easy to monitor X10 places and make the places share the configuration data with each other. It meets all the requirements of X10-FT for reliable failure detection, so we directly use ZooKeeper for convenience.

We build separate directories for each running X10 program in ZooKeeper, which are identified by their starting time. The architecture of the directories is shown in Fig. 4. The subdirectory named "places" under each program directory contains a few *ephemeral* nodes, which are registered by every alive place. A place failure will be detected after a few unanswered heartbeat messages and result in an automatic removal of that place's corresponding node. The contents in each ephemeral node are the IP address and the listening port of the corresponding place. Since every place watches the changes of the "places" directory, every other place will be notified of the collapse of a place through regular heartbeat messages. This event will make them take necessary actions to respond to failures. When a rebuilt place reconnects to ZooKeeper, the updated IP address and listening port are saved in the ephemeral node newly registered, so that other places can successfully recover the connections between them and the rebuilt place using this information.

5.3. Place rebuilding

As mentioned before, an X10 place consists of two processes: the launcher and runtime. When a place crashes, all other places will get a notification. Among them, the parent place of the crashed one will take actions to rebuild a new Launcher on an available machine; while the other places will suspend their communications (especially assignment of new tasks) with the crashed one, and wait for its recovery. The new launcher initiates a connection to its parent and registers a new ephemeral node in ZooKeeper. After that, its previous child nodes will be notified and reconnect to it. Meanwhile, the new launcher will also create a new runtime. At this time, the connection requests among all launchers can be delivered successfully through the recovered tree architecture. The recovered tree architecture also facilitates the rebuilding of the runtime connections to the new place.

The new runtime should be initialized with a few modifications. Generally, initializing an X10 runtime includes several stages: initializing request handlers, building links between itself and the other runtimes, broadcasting its own configuration, registering additional function handlers and deploying an execution environment of the user program according to initial messages received from the main place, place 0. When an X10 program starts, the main place will initialize all places simultaneously. Hence, there are barriers among all places to synchronize their execution progress during the original initialization. However, when rebuilding a new runtime after a failure, the new runtime should not wait on those barriers, because it is the only one that needs to do the initialization and the other runtimes have already passed those barriers. So we need to make the new one cross these barriers automatically during rebuilding.

The rebuilt place is forced to send requests to others to fetch their configurations, and broadcast its own configuration simultaneously. It will also send requests to the main place to fetch the initial messages, and then try to recover its memory states from the checkpoint record. After that, it will send requests to every other runtime to get the tasks lost since that checkpoint. To achieve this, we need to keep messages sent in each place. Meanwhile, the initial messages that are used for initializing the execution environment should be always kept in the main place. Any operations of sending new tasks to the rebuilt runtime in other places, will be suspended until the initialization of the new runtime is completed. In other words, before the rebuilt runtime gets all the lost tasks, other runtimes can only send simple reply messages as required, even though the connections between itself and others have already been recovered. In order to reduce the unnecessary waiting time, if the rebuilt runtime luckily receives an "exit" message from the main place during its recovery, which means that the whole application actually is going to terminate before its crash, it will abort the recovery activity and terminate its

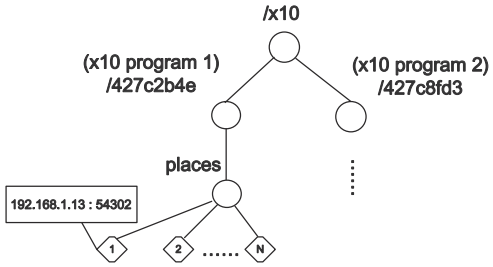

Fig. 4. ZooKeeper data structure.

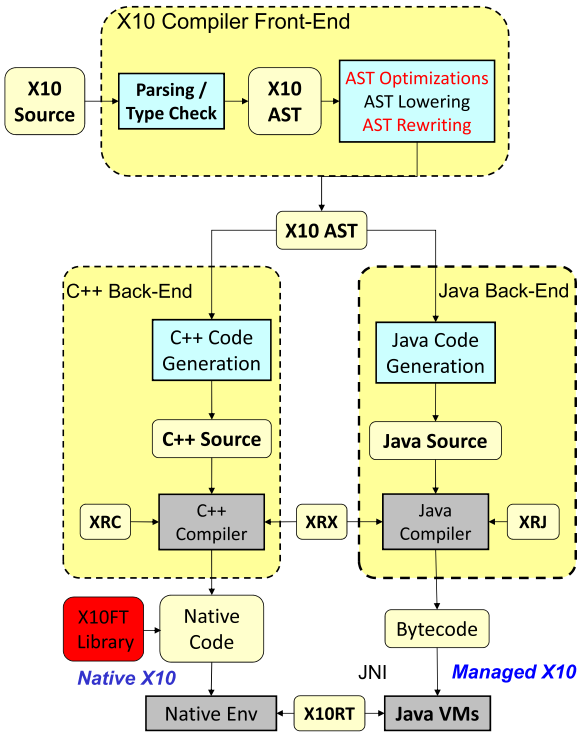

Fig. 5. X10 framework.

execution immediately. In the current prototype, when a failure of a place is detected, its parent place just rebuilds it on the machine where the parent resides. Because this machine is obviously available. The recovery makes an additional place, namely, the failed place, running on that machine. Usually it won't burden that machine heavily. We will use a sophisticated strategy to make the load of each place more balanced after recovery of failures.

5.4. Checkpointing

Fig. 5 illustrates the X10 framework. When compiling programs written in X10, the X10 compiler front-end will first do the parsing and type checking job, then it will produce the X10 AST, do the AST optimizations and lowering, and produce the canonical AST. Then, according to the different backend (Java or C++), X10 compiler does the corresponding code generation job (Java or C++), outputs the source code, and invokes the existing mature compiler, for example, the Oracle hotspot JDK or GNU GCC, to compile the generated source code. Meanwhile, additional X10 modules will also be linked into the program to generate the native executable or java bytecode. At run time, the binary will be loaded and run as a normal C++ or Java program with the precompiled X10RT library.

Since currently we only focus on the X10 C++ backend due to performance considerations, we have just modified the XRX (X10 Runtime in X10) module, the XRC (X10 Runtime in C++) module, and the X10RT (X10 Runtime) module as shown in the Fig. 5. We also modified the C++ code generation stage of the X10 compiler to directly instrument the generated C++ source files. We provides a library written in C++ (the red box[1] in Fig. 5) for the X10 runtime to interact with the underlying HDFS and do the fault tolerance job. This library mainly consists of interfaces interacting with the HDFS. We also added extra passes in the AST optimization stage shown in Fig. 5, to do liveness analysis on the X10 AST and rewrite the AST to make the program automatically save live data into HDFS. The liveness analysis pass performs classic dataflow equations on the CFG constructed from X10 AST of a program to find the live local variables at each *finish{}* checkpoint. The global variables in each place are identified by searching the invocation points of relevant runtime functions provided in X10.

5.5. Recovery

Sentinels: In X10-FT, each place needs to maintain information of the execution constraints of all tasks sent from itself, in order to make the recovery consistent in case of a place failure. There are two kinds of "sentinels" in the transmitted-message queue in a place, which introduce two constraints of the execution order of tasks during recovery. One is the "local sen-

[1] For interpretation of color in Fig. 5, the reader is referred to the web version of this article.

tinel", which corresponds to a local barrier inside a place, for example, an internal *finish{}* statement inside an *at (p) {}*. The other is the "global sentinel", which corresponds to a global barrier among all places during the execution of a place, for example, the root *finish{}* statement in the main place. During recovery, the *Constraints* are as follows: (a) The tasks received from a place after a "local sentinel" must wait for the completion of the tasks from the same place before that "local sentinel". (b) The tasks received from all places after a "global sentinel" must wait for the completion of all the tasks from all places before that "global sentinel". These two constraints are consistent with the semantics of the corresponding barriers. For example, if during recovery, place j received a few messages from place i and k, then the task queue in place j would look like the following:

place i: i1 L i2 L i3 i4 G i5 i6

place k: k1 k2 L k3 k4 k5 k6 G k7

The "L" means a local barrier in corresponding place. While the "G" means a global barrier in corresponding place. Then the task i2 must be executed after i1 because of the "L" between them. k1, k2 could be executed simultaneously. i1, k1, k2 or i2, k1, k2 could also be executed simultaneously. Although i5, i6, k7 could be executed together, they can only be executed after all the tasks of all places before the same "G" finished, namely, i1–i4, k1–k6. During recovery, the rebuilt place will sync all the messages lost since the checkpoint from all other places. Then the enhanced X10 runtime will execute these tasks in the order constrained by these sentinels. These sentinels make sure that the recovery is correct.

Execution recovery: In our design, when a task takes a long time or launches new tasks into other places, we should record the execution flow to save time when doing recovery and make sure there are no side effects. Hence, we insert a few flags into X10 AST during compilation, to generate additional code in the "Code Generation" stage to automatically record and recover the control flow using labels.

Fig. 6 shows a possible implementation of the WordCount program, in which the remote activity task may take a long time. After compilation, besides the code dealing with checkpoints, X10-FT compiler also inserts code to generate labels before method calls and branches in the program code. At run time, these labels will be generated along the call path to mark the execution flow. When the program invokes a method or takes a branch, the corresponding label information is recorded, including its sequence number and input arguments. At each checkpoint, we can get a label sequence. After reaching a checkpoint, X10-FT stores the recorded information needed to rebuild this task, namely, intermediate data and the label sequence along the execution path, into HDFS. The labels are layered, which means that the sequence number of labels starts from scratch in each layer. Each label in the label sequence queue points to the current control flow in that layer during execution. For example, in the WordCount code shown in Fig. 6, checkpoint1 will record a label queue {seq1, seq2}, while checkpoint2 will record a queue consisting of {seq2, seq2}. The first "seq1" in {seq1, seq2} means the control flow has reached the "maps" function but has not arrived at the "reduces" function in the "at (p)" statement, which is in the first layer. The second "seq2" means the control flow has reached *checkpoint1* in the "maps" function, which is in the second layer. The label queue {seq2, seq2} of checkpoint 2 has the similar meaning.

During recovery, we first rebuild the task instance, then load the label sequence together with input arguments into memory and run the task. During re-execution, the enhanced X10-FT runtime will change the control flow of user code to skip executed code according to the sequence number recorded in the label queue. For example, in the above example, if the program crashes after checkpoint2, then during recovery, the program will skip the "maps" function and enter the "reduces" function because the first label in the label queue is "seq2". Through applying same input arguments when invoking a function during recovery, the control flow can arrive at the same checkpoint. Then the runtime will begin to recover the intermediate data recorded at that checkpoint. In current prototype, labels are automatically inserted before the *finish{}* structure and *at (p){}* structure. However, re-execution could also skip most code blocks, which usually spawn multiple child activities inside. We will further improve the prototype in the future.

At present, only the *at (p){}* language construct and the root *finish{}* construct in each *at (p){}* construct are considered for checkpointing in X10-FT. The other language constructs and synchronization primitives, such as the *collecting-finish* or the *clocks*, will be our future work. X10-FT only records *consistent* checkpoints , either explicit or implicit, in each place, in order to reduce the overhead. This means that the checkpoints will only be saved when there are no other tasks running at that time, which is checked by inspecting the status of the task pool of that place. The implicit checkpoints of a place are the points when a task finishes in that place at run time. The explicit checkpoints are located in the X10 source code of

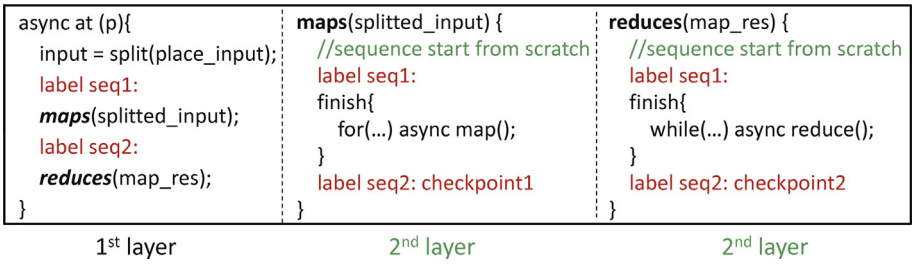

Fig. 6. A possible implementation of WordCount program.

applications by the compiler automatically. Based on our observations, there are only a few *consistent* implicit checkpoints when running X10 applications.

6. Evaluation

This section presents the evaluation of the performance overhead of adding fault-tolerance support to X10. We evaluate four benchmarks on a small cluster with 6 servers. These benchmarks are used to dissect the performance overhead of X10-FT. They represent different execution characteristics of X10 programs and have different kinds of checkpoint records. The WordCount benchmark is written in X10 according to the WordCount algorithm presented in the MapReduce paper [4]. The other three benchmarks are the SSCA#1 benchmark, which does bioinformatics optimal pattern matching; the Global RandomAccess (GRA) benchmark, which measures the rate of integer random updates of memory in the cluster; and the STREAM benchmark, which is a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernel. All these three benchmarks are provided in the official release of X10 2.2.1. We use the typical deployment configuration of X10 programs for each benchmark, that is to say, we deploy only one place on each node in the cluster. Since there are 6 servers, there will be 6 places for each benchmark, except for the GRA benchmark, which has 4 places, as it requires the number of places to be a power of 2. The number of worker threads in the worker pool of each place is determined by the number of CPU cores in that node.

The server configuration and software configuration in our evaluation are as follows:

- Server: Dell PowerEdge R715, 24 cores, AMD Opteron 6168, 1900 MHZ, 64G Memory, 2T Disk. The small cluster consists of 6 servers.
- Software: Debian squeeze 64bit, kernel 3.2.7. The X10 version is 2.2.1. The HDFS version is 0.20.203.0. The Zookeeper version is 3.3.5.

6.1. Failure recovery evaluation

We use manual fault injection to evaluate the capability of X10-FT to recover from possible failures. To get a thorough understanding of the fault-tolerance capability, we inject failures at 3 different positions in the source code. That is "ft-begin", "ft-middle", and "ft-end", which correspond to the front, middle, and end point in the execution flow of a program respectively. There are no checkpoints when "ft-begin" failures happen. All the failures are injected at a non-leaf node chosen from the tree architecture of X10 places. This will make X10-FT recover more connections between the involved nodes compared with the case of injecting failures at a leaf node. After the end of the recovered program, we will validate its result with the original one. Each fault injection test in our evaluation has been done 10 times in a row, and each value shown in the evaluation is the average.

6.1.1. Number of checkpoints

As expected, X10-FT successfully recovers each program from failures in all cases. The first and second rows in Table 1 show the number of the explicit checkpoints extracted from the X10 source code of each application at compile time. This is done by the X10-FT compiler through static analysis. The actual number of checkpoints at run time may be different when running the fault-resilient applications, as the third and fourth rows show. It may increase notably if some checkpointing code snippets are inside a loop. It may also decrease if some checkpointing code is not reached during execution.

There are also some implicit checkpoints which are recorded by the X10-FT runtime when each message (not an at (p) task) has finished its execution and there are no other tasks running at that time. The 5th row shows the final number of checkpoints in our evaluation after some optimizations. There are mainly optimizations. The first one is to make X10-FT only record consistent checkpoints. The second one is achieved by adjusting the checkpoint intervals. In our evaluation, we set the

Table 1
The number of checkpoints of applications in each place. The "N" in the "WordCount" column is the number of the worker threads. The "r" in the "SSCA#1" column is a parameter passed from user, which indicates the number of repetitions the program should execute the scoring algorithm.

	Applications			
	WordCount	SSCA#1	RandomAccess	STREAM
# Of explicit checkpoints at compile time	3	4	3	1
At (p) checkpoints	1	4	3	1
root-finish checkpoints	2	0	0	0
# Of checkpoints at run time	$2+\lceil \log_2 N \rceil$	$2r+2$	4+(0∼10)	1
At (p) checkpoints	1	$2r+2$	4	1
root-finish checkpoints	$1+\lceil \log_2 N \rceil$	0	0	0
implicit checkpoints	0	0	0∼10	0
# Of checkpoints after opts	2∼4	1∼2	2∼3	1

interval to be at least 20 s, which will reduce the number of checkpoints significantly. The length of the checkpoint interval is a tradeoff. If the interval is short, then the total checkpoint overhead will be large because there are more checkpoints. In contrast, if the interval is long, then the program may have to redo a lot of useful work that has been done, when it encounters failures and restarts the execution from a previous checkpoint. The optimal interval should be determined according to the applications and their inputs.

In the following, we will describe the time and space overhead for each benchmark in these fault injection evaluations.

6.2. WordCount

We evaluate the WordCount benchmark with different size of inputs and different number of activities in each place. According to Fig. 3 in Section 3, three checkpoints are found automatically by the X10-FT compiler. In the runtime, if these checkpoints are consistent, they will be saved into the HDFS. The dataset of WordCount contains the 1G, 8G and 16G bytes word files from Phoenix MapReduce [40]. We run the WordCount benchmark with the same input and configuration (such as the number of worker threads) 10 times and record the average results. The "1pxa" in the following tables and figures means that there are "x" worker threads in each place.

6.2.1. Benefits over a complete restart

Fig. 10 shows the benefit of the fault tolerance using X10-FT with the WordCount benchmark, compared with just restarting the benchmark after a node crash. The input size is 16G bytes. There are six groups of bars in this figure. The first bar in each group shows the sum of the "crash" time and the "restart" time. The "crash" time is the execution time of WordCount from the beginning to the point when it encounters a failure and the "restart" time is the execution time of restarting the program with 5 nodes after a node has crashed. The second bar ("X10-FT") in each group shows the execution time of WordCount in X10-FT when it encounters the same failure. The ratio of the X10-FT time to the crash-restart time in each group is 93.7%, 50.7%, 51.1%, 72.0%, 57.1%, and 54.6%, respectively. In the original version of X10 2.2.1, the program will get stuck permanently after a node crashed, unless a programmer finds that situation. In this evaluation, we assume that the programmer will find a place has crashed by checking the output of errors printed right before the failure and the program will get aborted after a place crash.

The file-reading stage of the WordCount program has been significantly optimized [5] compared with the original one using the native file operation interfaces in X10. However, even with such optimizations, the execution time of using X10-FT in the "ft-middle" case and in the 1p4a configuration, which means there are 4 workers in each place, still only takes 57% of the time of the naive crash-restart solution. It seems counter-intuitive that the "ft-end" has nearly the same execution time saved compared with that in the "ft-middle". This is because the interval between "ft-middle" and "ft-end" failures in run-time is short in this case, although their positions are far from each other in the source code. Nevertheless, in normal cases, the longer a program executes, the more time can be saved using X10-FT.

6.2.2. Space overhead

Table 2 shows the typical space overhead when taking a checkpoint record in the WordCount benchmark with different worker threads in each place, which mainly consists of the live local data at that checkpoint. As the number of worker threads increases, the size of the live local data also increases in most of the checkpoints in WordCount. The sizes of the total messages among the original X10 places are almost identical in the WordCount benchmark, despite the different input size and the different number of working threads in each place, so we only provide the results without giving the inputs and configurations, as Table 3 shows. In the fault-tolerant cases, there are slightly more messages than the original. This is because X10-FT needs to re-send the lost messages to the crashed place during the recovery, but they do not need to be re-saved in the sending place.

6.2.3. Time overhead

Figs. 7 and 8 show the execution time of WordCount in different test cases. The x-axis is the input size. The y-axis is the time spent in seconds. We can see from these two figures that the record overhead is negligible (1% on average) for the 1p1a case and modest (22% on average) for the 1p4a case compared with the "orig" time. There are mainly two reasons for this. The first one is that in WordCount the size of the live local data increases with the number of worker threads, so X10-FT needs to record more data in the 1p4a case. The second one is that there are more checkpoints as the number of worker threads increases in the WordCount benchmark, for example, there are 2 checkpoints in the 1p1a case, but 4 checkpoints in the 1p4a case. At each checkpoint, X10-FT needs to record all the live local data. We also can see from these two figures

Table 2
Size of local variables in a typical checkpoint record of WordCount.

Configuration	Size of local data (bytes)
1p1a	1,259,722
1p4a	5,038,758

Table 3
Messages overhead in WordCount.

	Msgs in Orig X10		Msgs in X10-FT	
	Size (bytes)	#	Size (bytes)	#
Place 0	9,107	224	11,298	263
Other places	1,224,060	35	1,224,076	30

that the recovery overhead of the "ft-begin", "ft-middle" and "ft-end" cases in 1p1a16G, e.g., 5∼14 s and the recovery overhead of these three cases in 1p4a16G, e.g., 4∼20 s, is usually either similar with or much smaller than the total record over head, e.g., 5 s in 1p1a16G and 33 s in 1p4a16G, respectively, since the recovery in X10-FT only needs to recover the states from the most recent checkpoint record.

Fig. 9 shows the breakdown of the record over head of a typical checkpoint record in the WordCount benchmark, the serialization time takes a great proportion (around 98%), because the structure of the live local data recorded is complicated. The time of HDFS-writing operations is negligible (around 1%) because the size of the live local data is small. Fig. 11 shows the breakdown of the total recovery overhead. We can see that the heartbeat-detecting time is between 9 s∼12 s, in spite of the fluctuation of the moments when failures happen in the heartbeat interval and the varying workload in each place when detecting failures. The same is true for the time of the "others" part (around 3∼5 s). The "checkpoint-recovering" time (around 7∼8 s) is basically the same as the sum of "HDFS-writing" time and the "serialization" time, because X10-FT needs to do the reverse job during recovery, that is, reading the same checkpoint record from the HDFS and then doing a deserialization.

6.3. Global RandomAccess

The Global RandomAccess (GRA) benchmark has two parameters, one is the size of the Global Array allocated in each place, the other is the number of updates to each element in that array. The total number of updates in each place is the product of the "updates" parameter and the size of Global Array in that place. These updates are sent as "remoteXor" messages between places. In this evaluation, we evaluate the GRA benchmark with the modest parameters that the servers can bear. The maximal size of Global Array in each place is up to 128 M, while the largest number of updates to an element is up to 250. There are no *finish* checkpoints in the GRA benchmark.

6.3.1. Space overhead

Table 4 shows the typical space overhead of a checkpoint record in the GRA benchmark with different parameters, and the total space overhead of all messages in each place. We can see from this table that the sizes of messages increase linearly as expected with the increasing parameters. In GRA, the number of messages in place 0 is almost the same as that in other places, because there are so many "update" messages in each place. The size of the checkpoint records in each place is almost the same as the size of the Global Array in that place.

6.3.2. Time overhead

Figs. 12 and 13 show the execution time of GRA in different test cases. We inject a "ft-middle" failure in the GRA evaluations. Most of the record overhead in each case (31∼361 s) is spent on recording the huge number of messages in the X10-FT runtime process. The recovery overhead in each case (12∼45 s) is much less than the corresponding record overhead, because generally in GRA, X10-FT only needs to read and deserialize the global array in the checkpoint record during recovery, whose size is moderate (128 M at most) compared with the size of messages. Fig. 14 shows the breakdown of the record overhead of a typical checkpoint record in GRA. Since the structure of the global array is simple, the serialization time in each case (30∼300 ms) is much smaller than the corresponding HDFS-writing time of states (920∼3500 ms). Fig. 15 shows the breakdown of the total recovery overhead. We can see that the time spent recovering the global data, namely, the "checkpoint-recovering" part, is reasonable (1∼4 s) compared with the other parts of the recovery in GRA. This is because the elements of an array, whose structure is simple, are serialized and deserialized faster than the live local objects in WordCount, whose structure is complex. The "detect" time in each case (8∼10 s) is still stable in GRA.

6.4. SSCA#1

The SSCA#1 benchmark matches two sequences by splitting the longest one into overlapped segments and concurrently comparing the segments with the shortest one in each place with a single thread. The user needs to provide a score matrix, which is used in the matching algorithm. The algorithm in SSCA#1 uses the dynamic programming techniques, that is, it trades space for time. There are two global arrays in each place of SSCA#1. The size of the larger array in each place equals the product of the length of the short sequence and the length of the long sequence in that place, which makes the size of the larger global array grow rapidly as the parameters increase. In addition, since the sizes of the sample long sequences provided in the official X10 2.2.1 release are small, we increase their sizes manually by appending each sequence to its tail a

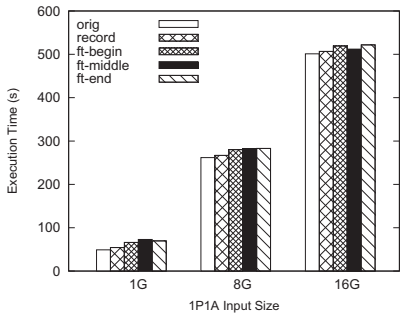

Fig. 7. Execution time of 1p1a.

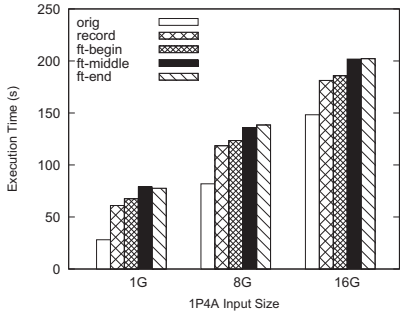

Fig. 8. Execution time of 1p4a.

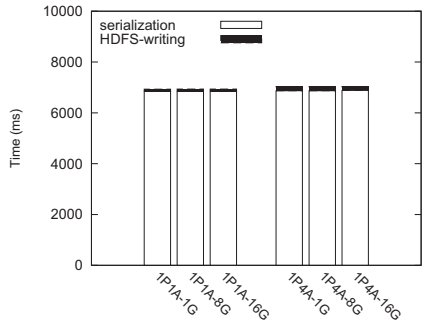

Fig. 9. Breakdown of record overhead.

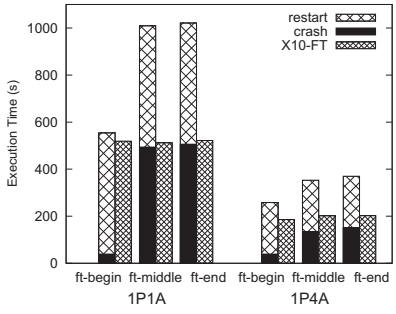

Fig. 10. Benefits of X10-FT compared with crash-restart in WordCount.

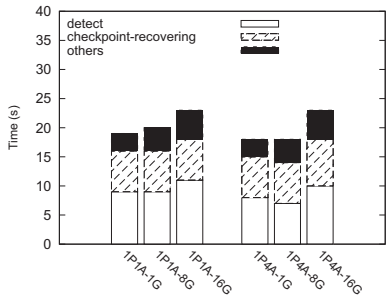

Fig. 11. Breakdown of recovery overhead.

Table 4
Space overhead of checkpoints & msgs in GRA.

Array size (bytes)	Updates	Checkpoint overhead	Msg overhead	
			Size (bytes)	#
256 K	50	256.07 KB	39,363,900	2,460,245
256 K	100	256.07 KB	78,771,592	4,923,228
256 K	150	256.07 KB	118,015,284	7,375,956
256 K	200	256.07 KB	157,470,088	9,841,884
256 K	250	256.07 KB	196,692,284	12,293,269
16 M	1	16 MB	50,383,164	3,148,949
32 M	1	32 MB	100,813,192	6,300,828
64 M	1	64 MB	201,514,376	12,594,652
128 M	1	128 MB	402,820,404	25,176,276

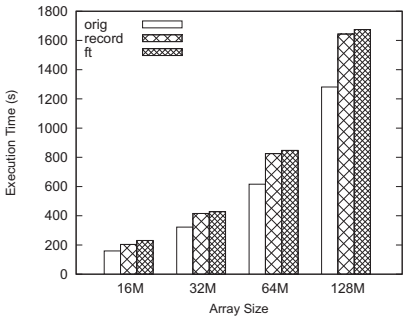

Fig. 12. Execution time of GRA with varying array size.

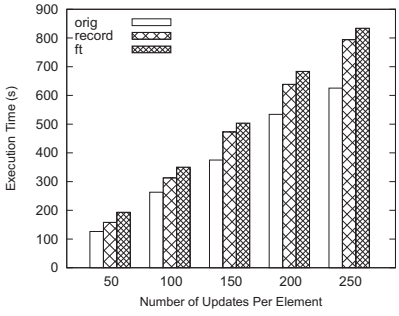

Fig. 13. Execution time of GRA with varying updates.

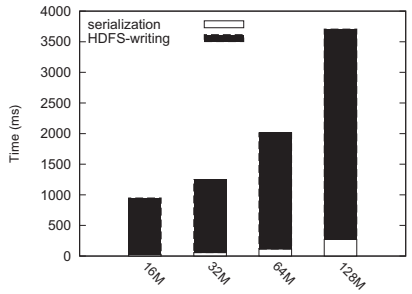

Fig. 14. Breakdown of record overhead.

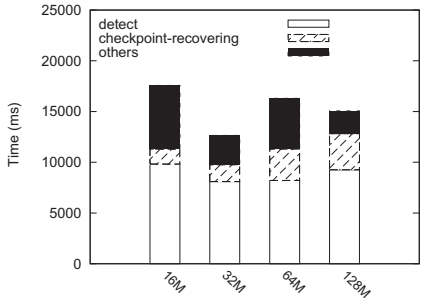

Fig. 15. Breakdown of recovery overhead.

few times. After the augmentation, now the sizes of the long sequences are 30 M, 40 M, and 50 M, respectively, while the size of the short sequence is 202 bytes. The scoring parameter is the default base-DNA matrix. We inject a "ft-middle" failure in the SSCA#1 benchmark. There is also no *finish* checkpoints in the SSCA#1 benchmark.

6.4.1. Space overhead

Table 5 shows the typical space overhead of a SSCA#1 checkpoint record, and the total space overhead of all messages in each place. The main part of the record is the larger global array. In SSCA#1, there are far more messages sent in place 0 than in other places, which is similar with the cases of WordCount and the following STREAM benchmark. This phenomenon indicates a general programming pattern in X10: the place 0 will distribute tasks to other places.

6.4.2. Time overhead

Fig. 16 shows the execution overhead of SSCA#1 in different test cases. Most of the record overhead in each case (21∼35 s) is spent recording the enormous global array in its place. The recovery overhead in each case (26∼32 s) is almost identical with the record overhead in SSCA#1. This is obvious, because X10-FT also needs to recover the same global array after failures. Fig. 17 shows the breakdown of the record overhead of a typical checkpoint record in SSCA#1. The serialization time in each case (2∼4 s) is also much smaller than the corresponding time of HDFS-writing (15∼27 s), which is similar to the case in GRA. This is because the structure of the global array is also simple in SSCA#1, despite its size being much larger than that in GRA. Fig. 18 shows the breakdown of the total recovery overhead after the failure. We can see that the checkpoint-recovering time takes a great proportion of the whole time. The checkpoint-recovering reads the states from the HDFS into memory and does a deserialization, which is the reverse process of saving a record. Figs. 17 and 18 are good explanations to Fig. 16.

Table 5
Space overhead of checkpoints & msgs in SSCA#1.

Length of long sequence (bytes)	Checkpoint overhead	Msg overhead			
		Place 0		Other places	
		Size (bytes)	Num	Size (bytes)	Num
30 M	967.95 MB	25,016,696	284	440	32
40 M	1.26 GB	33,348,362	284	440	32
50 M	1.58 GB	41,680,029	284	440	32

6.5. STREAM

The STREAM benchmark has only one parameter, which indicates the size of the three local arrays residing in each place. This benchmark does a simple vector computation in each place using the three local arrays and measures the sustainable memory bandwidth. In the official source code, there is only one remote activity in each place, and there are no *root finish* statements in the remote activities. So the checkpoints will only be taken when each *at (p)* statement finishes its execution. At that moment, the three local arrays will not be recorded, because their scope is limited to that remote activity. Hence, the size of checkpoint records is small in STREAM. We inject two kinds of failure, "ft-begin" and "ft-end", into the STREAM benchmark. There is also a global barrier in the STREAM source code, which is a suitable place for checkpointing. X10-FT does not handle it yet.

6.5.1. Space overhead

Table 6 shows the space overhead of X10-FT in the STREAM benchmark. In STREAM, X10-FT only records the message queues and the control flow labels in a checkpoint record. This is because there are no global data and live local data when writing a checkpoint record in STREAM. The number of messages is also small, since there is only one remote activity in each place and it does not send tasks to others as in GRA.

6.5.2. Time overhead

Fig. 19 shows the execution time of STREAM in different test cases. We can see from this figure that the record overhead in each case (less than 2s) is negligible compared with the original execution time in the corresponding case (36∼68 s). This is because the size of a checkpoint record is tiny. In some cases, e.g. when the size of the three arrays is 12 GB, the average execution time of the "record" case shown in this figure is shorter than the average execution time of the "original" case. This is because in the original X10 2.2.1, the execution time of the STREAM benchmark fluctuates widely, while in X10-FT, the execution time is more stable. The recovery time of "ft-begin" takes a little longer than that of the "ft-end". This is because in the "ft-begin" case, X10-FT needs to re-initialize the three large local arrays in the recovered place and redo the computations. In other words, Fig. 19 shows that re-executing the remote activity takes longer in the "ft-begin" case (around 10 s) than in the "ft-end" case (around 6 s). The time of "others" shown in Fig. 21 also explains this phenomenon. Fig. 20 shows the breakdown of the record overhead of a typical checkpoint record in STREAM. Since there are no global data and local data in the checkpoint records, only the time of HDFS-writing (around 260 ms) is shown in this figure.

6.6. Summary and discussion

Sensitivity study of overhead: In summary, the record and recovery overhead in X10-FT is application-specific. In X10-FT, a typical checkpoint record, which is made by an activity in a place, consists of the following four parts: (1) messages in that place, (2) global data in that place, (3) local data in that remote activity, and (4) control flow labels in that remote activity. For the "at (p)" checkpoints, it only contains the 1st and 2nd parts. For the "finish" checkpoints, it could have all of 4 parts. The 2nd and 3rd parts need to do serialization to make them storable, while the 1st and 4th parts are directly saved. The size of the 3rd and 4th parts varies greatly at different checkpoints. While the 1st and 2nd parts are recorded incrementally. The time to save a checkpoint record depends on two factors, one is the serialization time, which is determined by the size and structure of the data of (2) and (3); the other is the time taken to flush the states into the HDFS, which is determined by the size of states, that is, all the 4 parts. Similarly, the time to recover the states in a checkpoint record mainly depends on the time of reading states from HDFS into memory and the deserialization time of objects. In addition, it needs some time to make the control flow reach the one in the checkpoint record.

The size of these states and the structure of the objects at checkpoints are application-specific, which makes the overhead application-specific. For example, the overhead grows rapidly in some benchmarks like the GRA benchmark and the SSCA#1 benchmark as shown in the evaluation. In GRA benchmark, this is due to the big size of the messages and global data. In

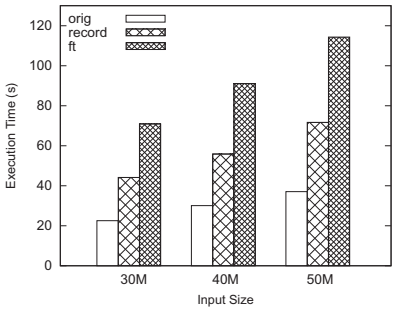

Fig. 16. Execution time of SSCA#1.

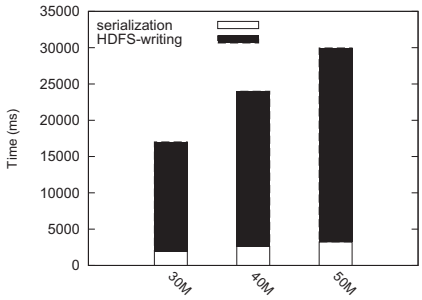

Fig. 17. Breakdown of record time.

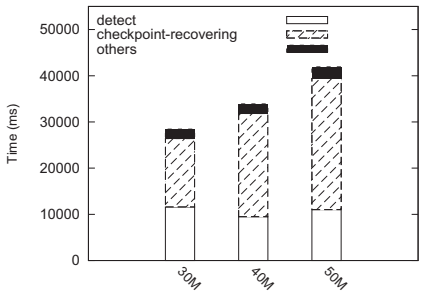

Fig. 18. Breakdown of recovery overhead.

Table 6
Space overhead of checkpoints & msgs in STREAM.

Size of three arrays in each place (bytes)	Checkpoint overhead	Msg overhead			
		Place 0		Other places	
		Size (bytes)	Num	Size (bytes)	Num
6G	0.05 KB	8,442	249	448	49
9G	0.05 KB	8,442	249	448	49
12G	0.05 KB	8,442	249	448	49

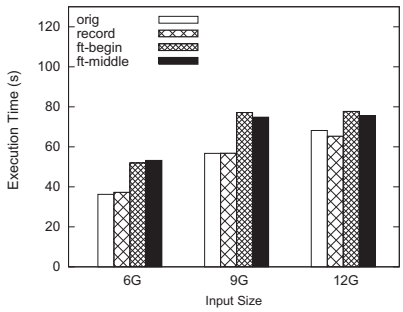

Fig. 19. Execution time of STREAM.

SSCA#1 benchmark, the overhead is due to the huge size of the global data generated by the matching algorithm based on dynamic programming. Although the size of these objects are big, the structure of them is simple. However, in WordCount, the size of objects is small, while the structure is complex. In addition, the size of the checkpoint records in the WordCount benchmark, which is a typical Map-Reduce application, is getting smaller and smaller as the execution progresses because of the ‘‘Reduce’’ feature. Hence, it’s better to checkpoint the states when the execution reaches the later ‘‘Reduce’’ stage. We can also see from the evaluation of WordCount that the size of local data may increase with the number of threads at some checkpoints. Some applications, like the STREAM benchmark, which has small number of messages, no global data, and

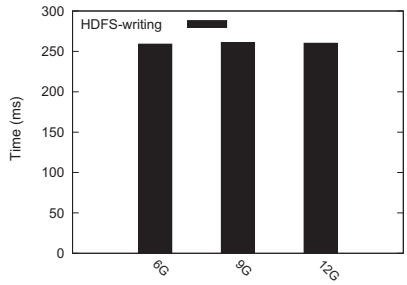

Fig. 20. Breakdown of record overhead.

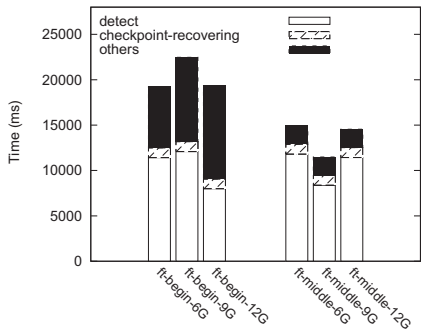

Fig. 21. Breakdown of recovery overhead.

no *root finish* statements in its remote activities, have very little overhead when doing fault tolerance using X10-FT as shown in the evaluation. In these applications, the remote activities in a place are simply re-executed if they have not finished their executions yet when that place crashes.

Generally, the overhead will increase with the size of states. The worst cases on the performance overhead of fault tolerance will happen when the size of each part is very large. But this situation rarely occurs in practice. In case that would happen, X10-FT has several ways to reduce the overhead. One way is to decrease the checkpointing interval. Another way is to record fewer kinds of checkpoints, such as not recording the "finish" checkpoint records inside a remote activity. The overall overhead could be reduced greatly by adjusting the checkpointing frequency in a proper way. In our evaluation, we choose the checkpointing interval to be at least 20 s after considering the execution time of all the benchmarks. Currently in the prototype, the frequency is not adaptable. We will make it adaptable at scale in the future. Meanwhile, if there are some global barrier points in the program, as we have mentioned in Section 4, all the checkpoint records before the recent global barrier point could be removed. Even in the worst case, when the overhead is not affordable, the user could also choose to use X10-FT to detect failures in time, which is also a great advantage compared with original X10.

Scalability: Due to the limited resources in the evaluation environment, we cannot evaluate X10-FT at scale. We just analyze its scalability in the following discussion. X10-FT consists of 3 major components, namely, ZooKeeper, DFS, and X10. So, intuitively, the scalability of X10-FT depends heavily on their scalability. All the three components have enterprise-class scalability. As the evaluations in the paper [41] shows, X10 itself has good scalability. The scalability of HDFS and ZooKeeper depends on the specific configuration of these systems. There are heartbeat messages between each X10 launcher, who acts as a ZooKeeper client, and a ZooKeeper server. Because ZooKeeper itself is distributed, which means that there are multiple servers, so there is no single point of failure in ZooKeeper and it should not be the scalability bottleneck. As for the HDFS, the evaluations in the paper [42] showed that the single namenode would become a bottleneck when the workload is write-dominated. In their evaluations, there were altogether 10,000 HDFS clients issuing write requests to a HDFS cluster consisting of 10,000 nodes, which makes the throughput of the namenode saturate. In X10-FT, each place will also write its own checkpoint records into the underlying HDFS independently. Hence, for some real HPC applications in X10-FT, whose scale is similar with that in their evaluations, the single namenode in the HDFS may become a bottleneck when huge number of places issuing write requests. There are several ways to improve it, such as distributing the namenode service. Meanwhile, in HPC clusters, a specialized storage network can also be used to improve the storage performance in X10-FT. These ways are orthogonal to X10-FT.

As for the overhead with X10 runs at scale, based on the above analysis, it also should not increase rapidly for each place. Because X10-FT itself is distributed, for example, the detection of failures is distributed, and each place saves its own checkpoint records. The number of messages in each place may increase linearly with the scale of places, while the others, such as the global variables, should not. Hence, the space and time overhead in each place may increase linearly with the number of

messages of linear growth. However, in many cases, the number of messages in each place, except place 0, is stable enough for X10-FT to scale well. The reason is that in these X10 programs, only place 0 will assign tasks to other places. So in such cases, the number of messages in other places won't increase with the scale of all places. Meanwhile, as place 0 is assumed to be always healthy in X10-FT, although it has more and more messages sent due to more places, its states won't be recorded. The optimizations mentioned, namely, adjusting checkpointing frequency and recording fewer kinds of checkpoints, can also be used to reduce the overhead for X10-FT to scale. The single namenode in HDFS, as we have mentioned, may increase the overhead at scale, especially when there are too many places. In such cases, the operations of saving checkpoints in each place may take longer time because of the contention.

7. Conclusion and future work

In this paper, we analyzed the language features and execution behaviors of a typical APGAS language, X10. Based on the analysis, we described the X10-FT framework that extends the APGAS model with fault tolerance support. We proposed a design to add fault-tolerance features to X10. Our design uses DFS and the Paxos model, along with record and recovery solutions based on concepts of *Place* and *Async* from the APGAS model.

We described our implementation of the X10-FT framework, which used ZooKeeper to detect failures, and then rebuilt the crashed places on another available node with the help of the enhanced runtime. X10-FT also modifies the X10 compiler to insert checkpointing code into user code during compilation. In order to durably store checkpoint data, X10-FT integrated HDFS as the persistent storage layer.

We have done a comprehensive evaluation using four practical benchmarks. The evaluation shows that X10-FT can successfully recover the program executions from place failures, with modest overhead in most cases. Even for the extreme benchmarks chosen from the HPC challenge benchmark suite and the SSCA suite, namely, the GRA, which uses a great amount of messages, and the SSCA#1, which needs enormous global data, X10-FT can still achieve fault-tolerance with acceptable overhead. The overall overhead could be further reduced using some optimizations, such as recording fewer kinds of checkpoints and adjusting the checkpoint frequency. Our experience suggests that providing fault tolerance is feasible, but requires non-trivial efforts. The evaluation indicates that the design of X10-FT picks a right direction to add fault tolerance to X10. X10-FT has made a first important step towards this direction.

The future work includes developing more sophisticated strategies to solve the load balance problem caused by failures, and adjusting the checkpointing frequency at scale. We also plan to address more issues, such as tackling the failures of the main place. A more solid implementation will be provided. The overhead and scalability of X10-FT will also be evaluated using more complicated applications on large-scale clusters.

References

[1] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, O. Tardieu, The asynchronous partitioned global address space model, in: Proc. First Workshop on Advances in Message Passing, 2010.
[2] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, D. Chavarría-Miranda, An evaluation of global address space languages: co-array fortran and unified parallel c, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2005, pp. 36–47.
[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005.
[4] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[5] C. Zhang, C. Xie, Z. Xiao, H. Chen, Evaluating the performance and scalability of mapreduce applications on x10, in: Advanced Parallel Processing Technologies, 2011, pp. 46–57.
[6] B. Schroeder, G.A. Gibson, Understanding failures in petascale computers, Journal of Physics: Conference Series, vol. 78, IOP Publishing, 2007, p. 012022.
[7] L. Lamport, Paxos made simple, ACM Sigact News 32 (4) (2001) 18–25.
[8] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.
[9] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, Zookeeper: wait-free coordination for internet-scale systems, in: Proc. 2010 USENIX Conference on USENIX Annual Technical Conference, vol. 8, 2010, pp. 11–11.
[10] D.A. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, A. Krishnamurthy, Designing scalable synthetic compact applications for benchmarking high productivity computing systems, Cyberinfrastructure Technol. Watch 2 (2006) 1–10.
[11] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, D. Takahashi, The hpc challenge (hpcc) benchmark suite, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Citeseer, 2006, p. 213.
[12] E.N. Elnozahy, L. Alvisi, Y.-M. Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. (CSUR) 34 (3) (2002) 375–408.
[13] K.-H. Huang, J.A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Trans. Comput. 100 (6) (1984) 518–528.
[14] G. Bosilca, R. Delmas, J. Dongarra, J. Langou, Algorithm-based fault tolerance applied to high performance computing, J. Parallel Distrib. Comput. 69 (4) (2009) 410–416.
[15] C. Engelmann, G.R. Vallee, T. Naughton, S.L. Scott, Proactive fault tolerance using preemptive migration, in: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2009, pp. 252–257.
[16] C. Wang, F. Mueller, C. Engelmann, S.L. Scott, Proactive process-level live migration in hpc environments, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, p. 43.
[17] J.S. Plank, K. Li, Faster checkpointing with n+1 parity, in: Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers, IEEE, 1994, pp. 288–297.
[18] J.S. Plank, K. Li, M.A. Puening, Diskless checkpointing, IEEE Trans. Parallel Distrib. Syst. 9 (10) (1998) 972–986.
[19] A. Beguelin, E. Seligman, P. Stephan, Application level fault tolerance in heterogeneous networks of workstations, J. Parallel Distrib. Comput. 43 (2) (1997) 147–155.

[20] C.-C. Li, W.K. Fuchs, Catch-compiler-assisted techniques for checkpointing, in: 20th International Symposium Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers, IEEE, 1990, pp. 74–81.
[21] Q. Gao, W. Yu, W. Huang, D.K. Panda, Application-transparent checkpoint/restart for mpi programs over infiniband, in: International Conference on Parallel Processing, 2006. ICPP 2006, IEEE, 2006, pp. 471–478.
[22] N Ali, S. Krishnamoorthy, N. Govind, B. Palmer, A redundant communication approach to scalable fault tolerance in pgas programming models, in: 2011 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2011, pp. 24–31.
[23] J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global arrays: a nonuniform memory access programming model for high-performance computers, J. Supercomput. 10 (2) (1996) 169–189.
[24] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Aprà, Advances, applications and performance of the global arrays shared memory programming toolkit, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 203–231.
[25] A. Vishnu, H. Van Dam, W. De Jong, P. Balaji, S. Song, Fault-tolerant communication runtime support for data-centric programming models, in: HPC, 2010.
[26] J. Nieplocha, B. Carpenter, Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems, Parallel Distrib. Process. (1999) 533–546.
[27] D.P. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D.M. Brown, J. Nieplocha, Transparent system-level migration of pgas applications using xen on infiniband, in: 2007 IEEE International Conference on Cluster Computing, IEEE, 2007, pp. 74–83.
[28] V. Tipparaju, M. Krishnan, B. Palmer, F. Petrini, J. Nieplocha, Towards fault resilient global arrays, Parallel Comput.: Architectures, Algorithms, Applications 38 (2007) 339–345.
[29] J. Dinan, A. Singri, P. Sadayappan, S. Krishnamoorthy, Selective recovery from failures in a task parallel programming model, in: Proc. 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010.
[30] G. Stellner, Cocheck: checkpointing and process migration for mpi, in: Proc. 10th International Parallel Processing Symposium, IPPS '96, 1996, pp. 526–531. http://dx.doi.org/10.1109/IPPS.1996.508106.
[31] G. Zheng, L. Shi, L.V. Kalé,, Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi, in: Proceedings of International Conference on Cluster Computing, IEEE, 2004.
[32] G. Fagg, J. Dongarra, Ft-mpi: fault tolerant mpi, supporting dynamic applications in a dynamic world, in: Recent Advances in Parallel Virtual Machine and Message Passing, Interface, vol. 1908, 2000.
[33] A.B. Nagarajan, F. Mueller, C. Engelmann, S.L. Scott, Proactive fault tolerance for hpc with xen virtualization, in: Proceedings of the 21st Annual International Conference on Supercomputing, ACM, 2007.
[34] H. Chen, R. Chen, F. Zhang, B. Zang, P. Yew, Mercury: combining performance with dependability using self-virtualization, in: International Conference on Parallel Processing, IEEE, 2007.
[35] J. Abawajy, Fault-tolerant scheduling policy for grid computing systems, in: Proc. 18th, International, Parallel and Distributed Processing Symposium, 2004. p. 238.
[36] M. Castro, B. Liskov, Practical byzantine fault tolerance and proactive recovery, ACM Trans. Comput. Syst. 20 (4) (2002) 398–461.
[37] J. Leon, A.L. Fisher, P. Steenkiste, Fail-safe pvm: a portable package for distributed programming with transparent recovery, Tech. rep., 1993.
[38] A. Agbaria, R. Friedman, Starfish: fault-tolerant dynamic mpi programs on clusters of workstations, in: Proc. Eighth International Symposium on High Performance Distributed Computing, 1999, pp. 167–176.
[39] G. Altekar, I. Stoica, Odr: output-deterministic replay for multicore debugging, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 193–206.
[40] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating mapreduce for multi-core and multiprocessor systems, in: HPCA, IEEE, 2007.
[41] O. Tardieu, D. Grove, B. Bloom, D. Cunningham, B. Herta, X10 for productivity and performance at scale, in: A Submission to the 2012 HPC Class II, Challenge, 2012.
[42] K.V. Shvachko, Hdfs scalability: the limits to growth, Login 35 (2) (2010) 6–16.