X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime

Chenning Xie† Zhijun Hao ‡ †
Haibo Chen †
† Institute of Parallel and Distributed Systems, School of Software, Shanghai Jiao Tong University
‡ School of Computer Science, Fudan University
xiechenny@sjtu.edu.cn, haozhijun@fudan.edu.cn, haibochen@sjtu.edu.cn

ABSTRACT

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 productivities as threaded and distributed programming is hard to use correctly and concurrency/distributed bugs are hard to spot. Asynchronous partitioned global address space (APGAS) model is a programming model aiming at unifying programming for multicore and clusters at good productivity. Unfortunately, the current implementation of APGAS programming model lacks support for fault tolerance and a single transient failure may render hours to months of computation useless.

In this paper, we make the first attempt to add fault tolerance support to APGAS programming models by integrating advances in fault-tolerant distributed systems to an APGAS language called X10. We thoroughly analyze the feasibility of providing fault tolerance for X10. Based on the analysis, we design and implement a fault-tolerance framework called X10-FT that leverages advances 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, which allows transparently handling machine failures in different granularities. Using the features of the APGAS model, we extend the X10 compiler to automatically locate execution point to checkpoint program states without intervention from the user. We also provide a preliminary evaluation show the cost of providing fault-tolerance in X10-FT.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Languages, Performance, Design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PMAM '2013, February 23, 2013, Shenzhen [Guandong, China]
Copyright 2013 ACM 978-1-4503-1908-9/13/02 ...$15.00.

Keywords

X10, Asynchronous Partitioned Global Address Space (APGAS), Fault Tolerance

1. INTRODUCTION

The emergence of multicore machines and multicore-based clusters has made it vitally important to ease programming on such platforms. Usually, there are multiple yet likely conflicting factors that need to be considered when designing such a programming model, including performance, expressiveness, programmability, and fault tolerance.

An attempt to ease programming on both cluster and multicore machines is the asynchronous partitioned global address space (APGAS) model. APGAS model is an extension of PGAS. 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 concepts of *Place* and *Async*, to overcome two shortcomings: 1) requiring machines with similar hardware configuration and 2) cannot dynamically spawn multiple activities. A recent embodiment of APGAS model is the X10 language [3]. X10 hides users from underlying machine topology and allows users to conveniently write multi-threaded programs that can be executed in cluster environments. X10 uses places to abstract address spaces and allows users to spawn activities as the computation units to run on either their home place or remote places. A number of other programming models, including MapReduce [6], can be easily expressed in X10 [17].

Unfortunately, though the APGAS model has the potential of embracing both performance and productivity, there is currently no support of fault tolerance in known languages and runtime based on it. Providing fault tolerance is important as many current computation tasks usually require running several days or even months on several thousands of cores. Not considering fault tolerance may make even a small failure in a small component render the whole computation task meaningless, requiring a restart of the whole computation or even recurring restarting the tasks. With the increasing scale of machines, the potential error rate grows as well, which makes the problem even more serious.

In this paper, we make the first comprehensive analysis on how to provide fault tolerance to APGAS-based language and runtime, by using X10 as an example. The goal is to see how advances in distributed systems may help to pro-

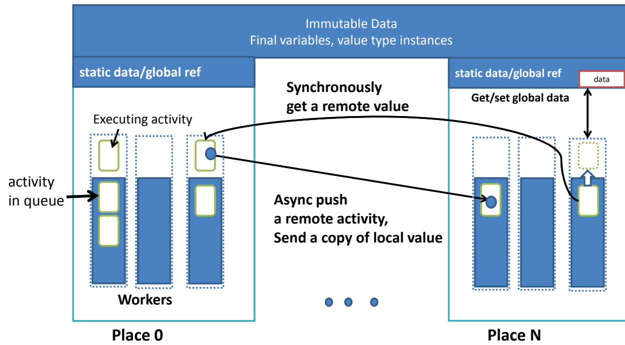

Figure 1: X10 Architecture

vide reliable and efficient fault-tolerance computation in the APGAS model.

As a typical APGAS programming model, X10 has some features that are related to fault tolerance. First, in each place, X10 maintains an exception-handling system similar to Java. Hence, we can mainly focus on fault tolerance of the remote access model between places. Second, the partitioned global address space model ensures that a global variable should only be accessed through a single agent place, which is responsible for the global variable during the whole execution. For other shared variables between tasks of different places, the language runtime helps to pass them through copies. Hence, it is possible to selectively redo activities by solving limited side effects. Besides, there is a set of explicit synchronization primitives, such as *finish*, *collecting-finish* and *at (P)*. These primitives help to switch execution flow between user code and runtime code, so that fault detection and recovery could be mainly implemented in the X10 runtime with very less modification to user code.

X10-FT leverages the language features of APGAS and combines them with advances in distributed systems for fault tolerance. Base on the key primitives such as *finish*, we modify the X10 compiler to automatically insert checkpoints into user code. X10-FT also implements an analysis phase in X10 compiler to derive a set of necessary intermediate data for recording in each checkpoint. Besides, users can also declare checkpoints with important intermediate data by themselves. To durably store checkpoints for further recovery, X10-FT seamlessly incorporates a distributed file system (DFS) to the language runtime of X10. X10-FT leverages the PAXOS [9] consensus protocol to reliably detect possible node failures or network partitions. When possible failures are detected, X10-FT resumes the disrupted activities by rebuilding the failure nodes (i.e., places), recovering the checkpoints from DFS, and changing the control flow to the latest valid checkpoint, to continue the execution.

Currently, we have done a preliminary implementation of X10-FT, including changes to X10 compiler, incorporation of the Hadoop distributed file systems (HDFS) [13] and ZooKepper [8] for consensus. This prototype can run simple programs with fault tolerance support. To evaluate the cost of providing fault tolerance in X10, we use WordCount [6], a typical application in distributed systems, and SSCA#1 [2], the bioinformatics optimal pattern matching that stresses integer and character operations, which is provided in X10 source code v2.2.1. We measure the performance under X10-FT. The evaluation result using different configurations show the overhead is modest.

The rest of this paper is organized as follows. Section 2 introduces some background information related to X10 and some necessary information regarding fault tolerance. Section 3 then presents the design of the X10-FT. Section 4 describes our current implementation status with some necessary supporting techniques to assist fault tolerance. Section 5 evaluates the performance overhead of providing fault tolerance in X10-FT. Finally, we discuss some related work and then conclude this paper with a brief remark on our future work.

2. BACKGROUND

In this section, we first present the two key idioms: *Place* and *Activity* in detail. We then illustrate programming using a APGAS-based language (i.e., X10) through an example - WordCount [6] written in X10. We also briefly discuss fault tolerance techniques in distributed systems and correlate them with language features in APGAS.

2.1 Place & Activity

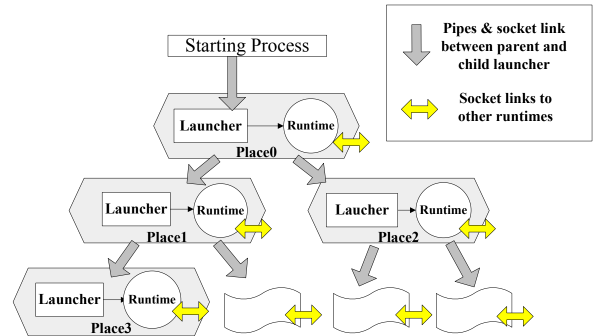

Figure 2: X10 Place Architecture

The address space model in X10 mainly uses the following idioms: *Place* and *Activity*. Places are independent entities in the model, and each place is an abstraction composed of two processes: launcher and runtime. User code will be wrapped into tasks called *Activity*, and be distributed to worker threads of different places partitioned on a cluster.

For a place abstraction, the launcher process is in charge of managing startup, exit, program output, and establishing socket links between runtimes. Launchers are built according to the structure shown in figure 2. In the full binary tree structure, parent nodes are responsible for forking launchers of child nodes. There are a communication socket and two pipes of standard output and error output between a parent and its children. The communication socket transfers the linking initial message to help building links between every pair of runtime, and control the exit of child nodes. The two pipes are used to make the output message flow through the tree structure and finally shown on the *stdout*. The runtime process provides the execution environment of the user code, and is responsible for the variable initialization, configuration broadcast, job tasks scheduling and so on.

Figure 1 illustrates the X10 architecture and the execution of X10 activities. Each place manages a pool, which contains some worker threads. According to the APGAS model, activities are allowed to be dynamically spawned and get executed by a worker thread synchronously or asynchronously.

The activity is called a *Remote Activity*, if it is pushed directly onto a new place from the other and independently executed. For instance, the main function of user code will be pushed onto the main place - place0, and the main function assigns tasks by pushing a new (remote) activity onto other places. The activities declared by the remote activities on the local place can be considered as child activities of the remote one.

In the APGAS memory model, all shared data is copied between remote activities from different places as shown in figure 1, except the GlobalRef data [12], which is in charged by an owner place. Hence, to access a GlobalRef data, a running activity should spawn a remote activity on the owner places. The remote activity helps to read or write the value, and copy the result back.

2.2 Key Primitives

Here, we use WordCount, a typical data-parallel application, as an example to illustrate the primitives in X10. The following is the pseudo code written in X10:

```
WordCount Pseudo Code:
 1  read(total_input);
 2  ###An X10 collecting-finish framework###
 3  total_result = finish(Reducer r) {
 4   for (p in Places) {
 5     place_input = split(total_input);
 6     async at(p) {
 7       read(place_input);
 8       finish for (i in numActivies) {
 9         act_input = split(place_input);
10         async {
11           ###count each word per activity###
12           act_result = map(act_input);
13         }
14       }
15       ###merge results between activities###
16       ###asynchronously###
17       place_result = reduce(act_result);
18       ###provide per place reduced result###
19       ###to collecting-finish framework###
20       offer(new ReduceArgs(place_result));
21     }
22   }
23  }###collecting-finish merge the results###
24  print(total_result);
```

- a) An ***async* {*S*}** is a typical statement in the APGAS model, which launches an activity and executes statement S without blocking the current activity.
- b) A ***finish S*** statement is a barrier that waits for all asynchronous activities spawned in S to terminate.
- c) An ***at (p) S*** statement appoints a specific place to execute statement S. The PGAS model partitions the shared address space among distributed machines, which are abstracted as places in X10. The communication mechanism between places is message passing.
- d) ***Collecting-finish***, a ***finish(r) {offer (Intermediate result T)}*** statement is the so-called collecting-finish framework in X10, which is very similar with the MapReduce [6] model. This framework is a typical combination of the statements mentioned above. It collects the intermediate results in each place, and automatically reduces them into the final result at Place0 by a user implementation of the Reducible interface.

Figure 3 shows the whole workflow and the internal states changes when running the WordCount application in the X10 runtime. First, the Place0 reads the input data, then splits and dispatches the data to independent places (0...N-1) to start a remote activity. The remote activity splits the data in each place and dispatches them to concurrent activities. The finish barrier ensures the completion of each map task, then the reduce stage begins, which reads the intermediate data generated from the map stage and merge them into the final results. When each place has finished its reduce stage, the collecting-finish framework will do the reduce job between places automatically, and will give the correct WordCount result.

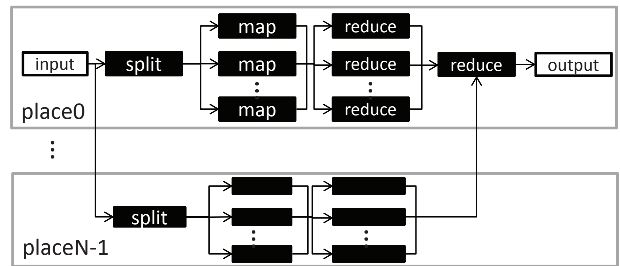

Figure 3: Word Count Execution Flow

2.3 Fault Tolerance in Distributed Systems

A distributed system is complex because of the communication and the consistency requirements caused by the geographical distances. 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: *computation replication* and *storage replication*. The former may have multiple same computation tasks executed at the same time, and the manager may choose a faster one or get a consistent result. It may cause loss of performance or require many more computing resources. Storage replication is also popular in many existing systems.

Existing distributed systems mostly have their own recovery model. For instance, the Hadoop MapReduce system restarts the mapper work or the reducer work if some nodes have problems. The benefits from its programming model that divides parallel computing into two explicit stages are a low coupling degree and no side effect. However, APGAS model provides a more powerful semantic support for user to freely construct a program. In spite of the explicit declaration to start new concurrent tasks, the diversity of the possible parallel structures becomes a challenge when supporting fault tolerance.

3. DESIGN

X10-FT adds fault tolerance to the APGAS model from several respects. First, it improves fault detection of remote accesses between places. It incoporates the PAXOS [9] pro-

tocol to reach consensus when there is a split-brain condition due to hardware failures or network partition. Second, X10-FT uses checkpoints to store computation states into the distributed file system for further recovery. Besides re-executing all the unfinished tasks of failed places, X10-FT also recovers inflight computation by taking checkpoints of running tasks at proper time. These checkpoints could be automatically set according to typical X10 structures such as *finish*{}, or manually set by user declarations.

X10-FT enables places to communicate with each other through PAXOS protocol to detect the failures, do the recovery when restarting a rebuilt place. It further provides a library for interfacing with the underlying distributed storage. All these are transparent to users.

At each proper timeline, X10-FT will capture states of places and the being performed tasks automatically, and write them into the underlying distributed storage. When there is a node crash or some other faults that cause the places on that node aborted, the PAXOS protocol with heartbeat detection helps to detect this event in time, then the parent place of the crashed one is notified to rebuild the crashed place on an available machine. The rebuilt place will load saved states into memory from the distributed storage, and resume the execution of tasks as captured in the checkpoint that recorded before the fault happened.

3.1 Failure Model

X10-FT is designed to recover fail-stop failures of any number of nodes (except the place0 node), without consideration of non-fail-stop errors such as the byzantine faults or defects in program itself. Further, we mainly focus on deterministic or idempotent workload, such as the scientific computation. It means that no matter how many times we execute the application workload, we will get the same final result if the input is the same. Most of them have few interactions with the outside world. We believe this assumption is reasonable, because the APGAS model is mainly used in the HPC application domain.

In our design, the parent place of a crashed place is responsible for detecting the failures, rebuilding a new place, and recovering the states of the crashed one. As in figure 2, the place0 is the root of the tree structure, which is usually in charge of controlling the other places' execution. In our present design, failures of the place0 are unrecoverable. This can be further fixed using a hot-standby technique that synchronizes states of place0 in two machines, which will be our future work. If multiple places crashed simultaneously, the parent of each one will rebuild them respectively.

We currently do not buffer I/O requests between checkpoints. Hence, if one task is rolled back to the checkpoint, users may observe some output printed again as that have done before place crashed. However, the output would be consistent.

3.2 Recording

X10-FT includes the modified X10 compiler and runtime. The compiler will do liveness analysis and automatically insert the recording code into the user-written X10 programs at the proper points according to features of the APGAS model. Meanwhile, the compiler will add labels to branches (including function calls) of the control flow of the program.

We make different places manage their activities separately. For each place, we record every new task (i.e., the remote activity including its child activities), and save the global variable states of places after this task completed. Inside a task, checkpoints are inserted during the sequential execution of the outermost activity. These checkpoints could be calculated automatically according to statements *async*, *finish* and *collecting-finish*, which could create and synchronize the concurrent child activities. During the checkpoint, all the intermediate data relevant to further computation will be saved.

Generally, different tasks on the same place should be executed independently. They only affect each other when access the same global variable managed. Hence, we maintain the write versions of a global variable, using sequence numbers with their modifiers. Each read access will get a version for dependency tracking. These versions will be recorded during the checkpoint, which helps to verify a happen-before consistency in recovery.

When current execution successfully launches a new task onto other places, X10-FT will add a record to avoid the new task to be duplicately declared in case that current execution rolls back to a previous checkpoint.

3.3 Recovery

Recovery of our X10-FT framework will take place when a collapse of places occurs, such as a node crash. For user code or worker thread errors, we leave them to the native exception system inside each place.

When a place collapses, a change of membership will be detected through the PAXOS protocol, which helps to get a new consistency among available places. These places then suspend communications with the error one. Parent node in tree structure of figure 2 is responsible to rebuild the crashed child place. After the failed places are rebuilt successfully, the new places participate in the PAXOS group and recover the communication links to others.

The rebuilt place will load unfinished tasks with all of their checkpoints from the storage. Tasks without a saved checkpoint are simply re-executed. The others will be recovered from a latest valid checkpoint after a consistency checking. In order to ensure the access consistency of global variables, for a checkpoint, we should get the version of each access to a global variable, and check whether the previous versions before this access have been saved into checkpoints of their modifiers. If so, it means that we get a consistent happen-before relationship in these saved checkpoints relevant to same global variables. Then we could recover to the current valid checkpoint. Otherwise, we will revert to the previous checkpoint which has less relevant versions of global variables and check again. In the worst case, it may be necessary to rerun the whole remote activity.

Fortunately, it rarely appears in APGAS model to deal with the worst case or to check a complicated version dependencies. Since this kind of version checking only happens if there are frequent alternate accesses from different concurrent tasks to the same global variables and meanwhile these tasks all have checkpoints inside.

4. PRELIMINARY IMPLEMENTATION

Currently, we have partially implemented X10-FT based on X10. The current version incorporates Zookeeper [8] to detect failures, and integrates HDFS [13] under the language layer of X10 to reliably store the checkpoints for recovery. We also implement the place rebuilding and modify X10

runtime as well as the X10 compiler to support a simple automatic control flow recording and recovery. For intermediate data recording and recovery, we currently have to manually modify the application source code written in X10 to interact with HDFS.

4.1 Incorporating HDFS

We have incorporated Hadoop distributed file (HDFS) into X10, since it is a relatively mature open-source distributed file system that has been widely used in many fields. We insert a set of interfaces relating to reading and writing distributed file system into X10 source code, and use the annotation provided in X10 to make X10 functions reference C++ APIs of HDFS. These functions help to implement the distributed file system interface, so that HDFS can be accessed from code of X10 runtime.

4.2 Integrating Zookeeper

Zookeeper is an open source tool that is widely used to monitor node failures in distributed system. Zookeeper extends the PAXOS protocol, implements heartbeat detection and small data sharing through replicas. These features make it easily adopted to monitor X10 places and share the place configuration data.

We build directories separately for each running X10 program, identified by a starting time. The data structure in the directory are showed in figure 4. The subdirectory named "node" is used to store the EPHEMERAL nodes, which are registered by every existing places. Any disconnection because of a place failure will result in an automatic removal of that place's corresponding node. The content of each node records the IP address and listening port of the corresponding place. Since each place watches the changes of "node" directory, the collapse of one place is notified to the others through regular heartbeat messages, making them choose appropriate recovery operations. When the rebuilt places relink to Zookeeper, IP address and listening port are also updated with registering a new EPHEMERAL node, so that other places can successfully get this information.

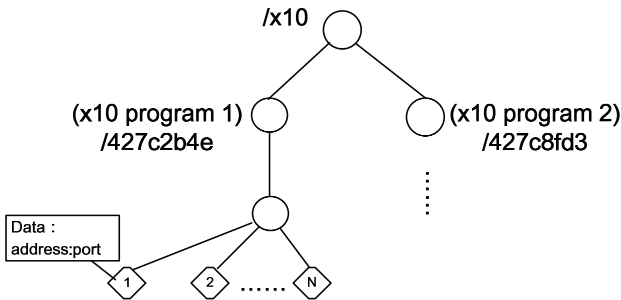

Figure 4: Zookeeper Data Structure

4.3 Place Rebuilding

As mentioned before, a place is an abstraction consisting of two processes: launcher and runtime, and we use Zookeeper to monitor the normal operations of each place. Thus, when the collapse of a process or machine causes a place crash, all the other places will get notifications. Among them, the parent node of the crashed place will take actions to rebuild a new Launcher on an available machine; while the other places suspend all the communication (especially new tasks assignment) with the crashed one, and wait for a connection recovery. The rebuilt launcher initiates a connection to its parent and registers a new EPHEMERAL node. After that, its previous child nodes will be notified and relink to the new launcher. Meanwhile, the rebuilt launcher will create a new runtime. Then the basic link request becomes available, so that other places can rebuild runtime connections to the new one.

The new runtime should be initialized with a little modification. Generally, establishing an X10 runtime includes several stages: initializing basic request handlers, building links between runtimes, broadcasting configuration, registering other function handlers and deploying user program environment according to initial messages sent from main place. There are barriers to synchronize all places to get the same stage consistently. However, when rebuilding runtime, we remove these barriers. The rebuilt one is forced to send requests to others to get all configurations, and broadcast self-configuration at the same time. It will also send requests to the main place to get initial messages, and then try to recover memory states and reload uncompleted tasks saved before. After that, it will send messages to every other runtime to get the job tasks missed since the crash. To implement this, we need retaining sent messages until target places complete stages saving data into disk, and the initial messages should be always retained by the main place. Any new tasks sent to the rebuilt runtime, such as starting remote activity and put/get remote values, will be suspended until the initialization is completed. In other words, before the rebuilt runtime gets all lost tasks, other runtime can only send basic reply as it required even through connection is recovered.

When the rebuilt place temporarily has no task communication with others, rebuilding executes silently without task suspending. To reduce unnecessary waiting time, if the rebuilt runtime receiving an exit message from the main place, it will abort recovery and finalize the execution immediately. In current implementation, when a place failure is detected, parent node just rebuilds the child place on local machine, which obviously is available. It is not a heavy burden when the crashes are rare. We will use a more optimized way with load balancing support in future.

4.4 Checkpointing

Figure 5 illustrates the X10 compilation procedure: For applications written in X10, the X10 compiler front-end will first do the parsing and type checking, produce the X10 AST, do the AST optimizations and lowering, and produce the canonical AST. Then, according to the different backend (Java or C++), the X10 compiler does the corresponding code generation (Java or C++), outputs the source code, and invokes the existing mature compiler (Oracle hotspot JDK or GNU GCC) to compile the generated source code. Meanwhile, the underlying library, such as X10 runtime and some other functional modules, will be linked to the program. This will produce the bytecode or the Native code for Java and C++ backend respectively. Then the binary will be loaded and run as the normal Java or C++ programs with the precompiled X10rt library in the system.

We have modified the XRX (X10 Runtime in X10) part and provide a *X10 Fault Tolerant Lib* library written in C++ (the red part in figure 5) for the X10 runtime to interact

with the HDFS and do the fault tolerance. This library is mainly for interfaces interacting with the HDFS. We will also modify the AST optimization stage shown in figure 5, to do a liveness analysis on the X10 AST and rewrite the AST to make the program automatically record live data into HDFS.

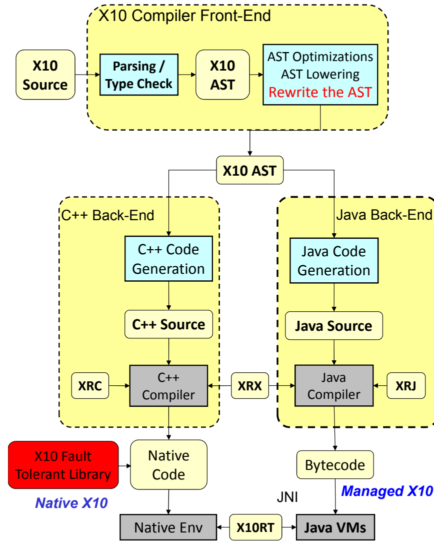

Figure 5: X10 Compilation Process

Execution Recovery: In our design, when a task takes a long time or launches other new tasks into other places, we should record the execution states to save time in recovery and make sure there is no side effect. Hence, we insert checkpoint flags into AST during compiling to export some extra code in generated C++ file.

The following code list is a possible implementation of the WordCount program, in which the remote activity task may take a long time. Besides extra codes dealing with checkpoints, we insert control labels before method calls and branches, which are in the call path to approach a checkpoint and will get a sequence number when executing. When calling a method or executing a branch, the corresponding label information is recorded, including its sequence number and input arguments. After reaching a checkpoint, we store the recorded information about rebuilding this task, intermediate data and the label information related to the call route into HDFS. In our example of following code list, checkpoint1 will record a sequence number queue {seq1, seq1}, while checkpoint2 will record {seq2, seq1}.

To recover execution, we first rebuild the task instance, and then load the control labels sequence together with input arguments into memory and run it. During re-execution, the X10 runtime is modified to change control flow of user code to skip executed code according to the sequence number queue. Combining with applying the same input in the calling route, we can get the checkpoint and recover the intermediate data.

In current implementation, checkpoints are automatically set according to *finish* structure, and re-execution could skip most code blocks with declaring multiple child activities inside. In future, we plan to add a more complicate solution to get an accurate call path to add checkpointing.

```
A Possible Implementation of WordCount Program:

at each Place: launch remote activity {
    split(input)
    label seq1: map result = map() {
            #inside map function
            finish {map}
            #seqnumber start from scratch
            #inside function
            label seq1: checkpoint1
    }
    label seq2: reduce result = resuce{
            #inside function
            finish {reduce}
            label seq1: checkpoint2
    }
}
```

5. EVALUATION

This section presents a preliminary evaluation of the performance overhead of adding fault tolerance to X10 in two levels: Activity level and Place level. At activity level, we mainly focus on the cases within one place. We record the intermediate data that are in the form of X10 objects during normal execution, inject a transient failure of the remote activity after a checkpoint, and recover the data from the checkpoint in HDFS. At place level, we provide a complete functional test of two applications, on both single machine and a small cluster with 4 machines. In this case, we kill one non-leaf node in tree structures of X10 places, and let the framework automatically recover from failure. The WordCount application will recover from the checkpoint, while the SSCA#1 benchmark will simply restart the failed task.

The WordCount benchmark is written by us according to the WordCount in the MapReduce paper [6]. The SSCA#1, a bioinformatics optimal pattern matching, is provided in the official release of X10 v2.2.1.

5.1 Activity Level Fault Recovery

In the activity level evaluation, our machine configuration is as follows:

- Hardware: Intel Core 2 Quad Processor Q9300 (6M Cache, 2.50GHz, 1333MHz FSB), 2G Memory, 1TB Disk.
- Software: Debian squeeze 64 bits, kernel 2.6. The X10 version is 2.2.1. The HDFS version is 0.20.203.0. The Zookeeper version is 3.3.5.

We test the cases with different number of activities in a single place . According to the WordCount Pseudo Code in section 2, we insert five checkpoints after line number: 3, 6, 14, 17 and 20. The test data set is the WordCount.x10 source file, whose size is 150KB and two randomly generated word files, with 10M and 1G bytes size respectively.

5.1.1 Test Methodology

We manually inject the data recording and recovering code at each checkpoint in the compiler generated c++ files. At each checkpoint, the program will first serialize all live objects at that point, and write the object stream into the HDFS. Then the objects will be read back from the HDFS

immediately and deserialized into memory. The execution continues using the deserialized objects. After the program finished, we will validate the result.

5.1.2 Space Overhead

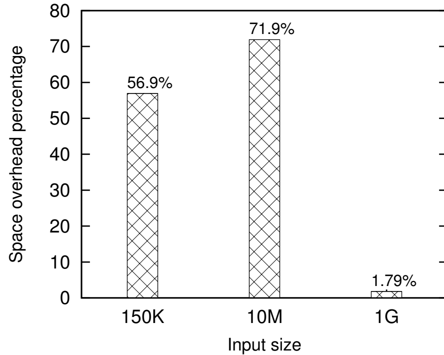

Figure 6: Space Overhead Relative To The Inputs

Figure 6 shows the space overhead for the three input sizes. The percentage is calculated as the total data size recorded during all checkpoints relative to the input size. In WordCount, the program takes the checkpoints for 5 times. We only list the result running with one place and one activity (1p1a). The results of other cases such as one place with 4 activities (1p4a) are similar. Actually, the space overhead in checkpointing in a place has almost no relationship with the X10 launcher options (1p1a or 1p4a).

As shown in figure 6, the space overhead becomes negligible as the input size grows larger. When the input size reaches 1GB, the total space overhead is just 18MB, which is only 1.8% of the input. This is mainly due to the *reducible* characteristics of WordCount. The overhead is also application specific. We believe that this bound will be further decreased as the places number increases (the workers number grows up, while the input data size for each places decreases). For the smaller input cases, the overhead is moderate, 56% and 71% for the 150K and 10M input respectively. We believe that applications with small input and short runtime may not fit X10-FT, as it usually does not require fault-tolerance support.

Figure 7 shows that the time that WordCount spends in the 1p1a and 1p4a cases, respectively. The runtime is composed of three parts: 1) the native part; which means the normal execution: 2) the Record part; which means the time spent on the checkpoint recording; and 3) the recover part, which means the time spent when doing the recovery after failure. The y-axis represents the absolute time, the 1p1a stands for 1 place with 1 activity, while 1p4a the 1 place with 4 activities.

As we can see from the figure 7, the native time decreases almost linearly as the number of worker increases from 1 to 4, while the record and recover parts are nearly constant. This is because most of the checkpoints are taken at the sequential part of execution by the main thread in each place, these two parts are not parallelized.

The record time and recover time for each running case are almost the same, because these time portions are only relative to the intermediate data size needed to be recorded and recovered.

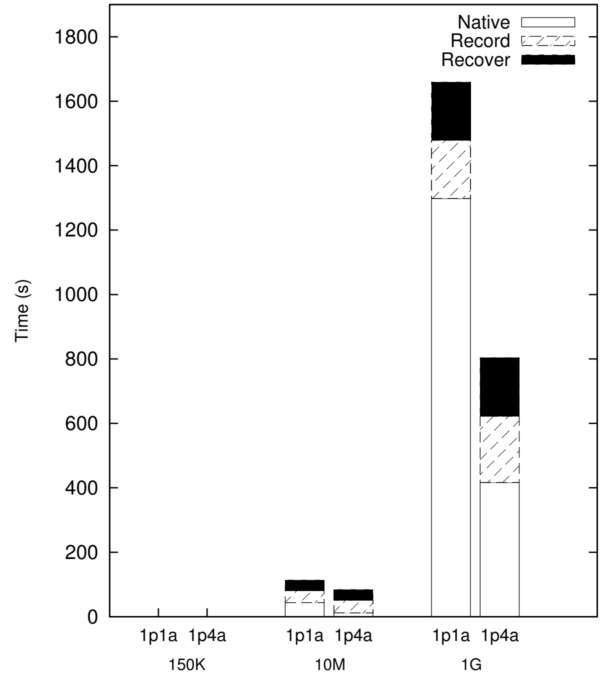

Figure 7: Time Overhead For Different Inputs

The time spent when the input is 150K is too small to be shown on the figure, but the situation is similar with the cases of 10M, 1G inputs.

Figure 8 shows the percent of the record time relative to the native time. In figure 8, the best case is the 1G input one. 13% of the native time for doing checkpoint is well enough. For the 150K input case, the record time is 360% relative to the native. For the 10M input case, the 1p4a case is obviously worse than the 1p1a case, because the native time could be speeded up as the number of worker thread increases, so the ratio will be higher. From figure 8, we can get the following insights: the X10-FT fault tolerant framework is more suitable for long-running, big input and small output applications. Such application are actually quite common for scientific applications.

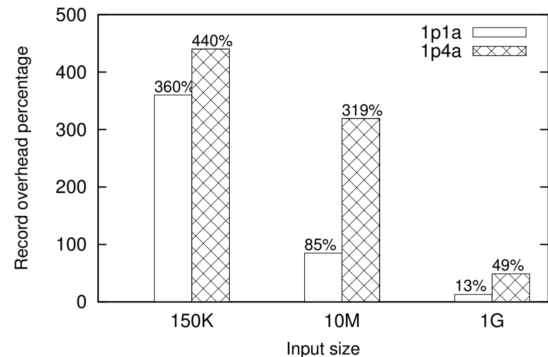

Figure 8: Record Time Overhead Relative To Native

Figure 9 shows the breakdown of the record time when running WordCount with 10M input size and 1p1a configuration. There are 5 bars in the figure, corresponding to

the 5 stages in WordCount execution. The native time is 45536 ms, and the total record time for 5 stages is 39027 ms. The record time is composed of two parts, the serialization time and the HDFS write time. The recovery time is similar with the record time, and is composed of the HDFS read time and the deserialization time. We can see from this figure that the serialization time takes most of the proportion in the total time. The two data labels of each bar correspond to the absolute time of each part. For example, in stage 3, the serialization part takes 11071 ms, while the HDFS write part only 31 ms. The bottleneck of the record procedure is the serialization protocol we used. Currently, we used the X10 native serialization protocol. The Google's Protocol Buffers may have better performance for doing serialization, which we will test in the future. For the multiple places case, there may be some contentions when they write and/or read HDFS simultaneously. The performance of our APGAS fault tolerant framework will be more dependent on that of the underlying HDFS.

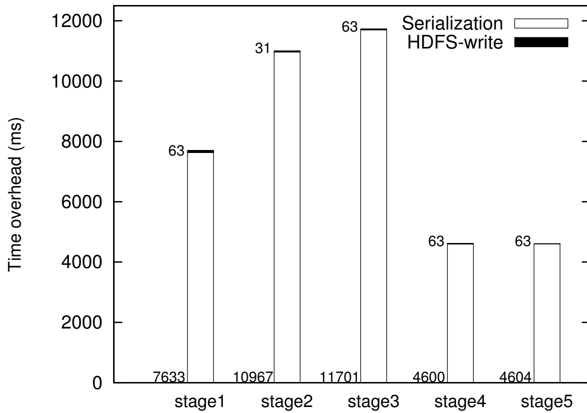

Figure 9: Record Time Breakdown for 10M input With 1P1A

5.2 Place Level Evaluation

In place level evaluation, our machine configuration is as follows:

- Hardware: one Dell PowerEdge R715 machine, 24 cores, AMD Opteron 6168, 1900MHZ, 64G Memory, 2T Disk. The small cluster consists of 4 machines.

- Software: Debian squeeze 64bit, kernel 2.6.38. The X10 version is 2.2.1. The HDFS version is 0.20.203.0. The Zookeeper version is 3.3.5.

In this test, we inject failures to make a non-leaf node of place crash, which has both parent and child nodes in the tree structure. We provide the complete functional evaluation through two different behavior of WordCount and SSCA#1 after the failure is detected. The recovery result is compared with a data-record-only execution using X10-FT without failures and a native x10 v2.2.1 execution.

5.2.1 WordCount

In the test of WordCount, we inject the failure where the remote activity on a place is about to be completed. The X10-FT framework will detect this failure, rebuild the crashed place, and then restart the incomplete activity in the rebuilt place. The socket links to other places are also reconstructed. The rebuilt place will skip most of the executed procedures to fast-forward to the checkpoint.

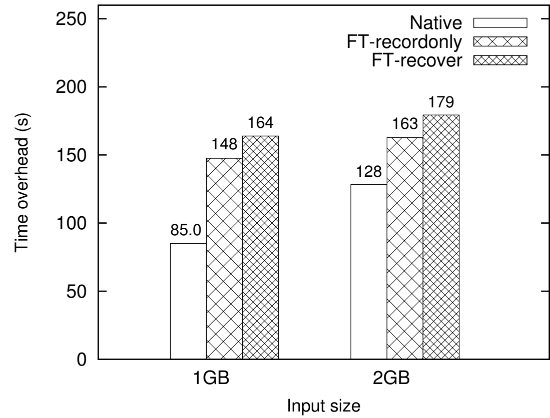

Figure 10: WordCount runs on single machine: 4places, 6 map tasks each place, input size: 1GB and 2GB

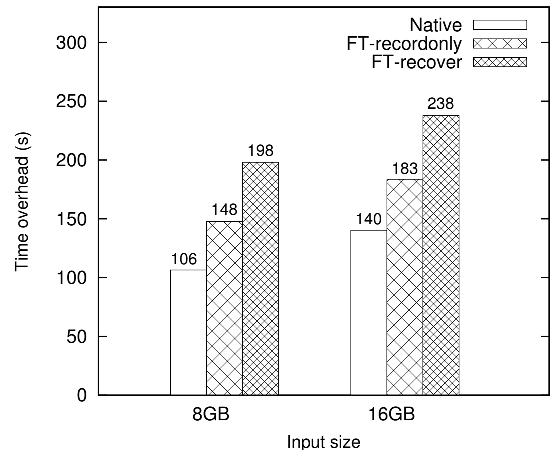

Figure 11: WordCount runs on cluster of 4 machines: 4 places, 20 map tasks each place, input size: 8GB and 16GB

Figure 10 and figure 11 show the WordCount result for the place level test. In fact, our framework successfully recovers the execution from failure and gets the correct result. Overhead in the record-only test includes heart beat monitored by Zookeeper, control flow and intermediate data recording, and the serialization of intermediate data during recording accounts for the major part. In addition to record-only overhead, recovery test contains an extra time slot of delay in failure detection (X10-FT confirms a failure when timeout), place rebuilding and recorded data recovering from DFS. Similarly, deserializing recorded data into runtime objects also accounts for a large proportion of recovery time. Comparing to running on single machine, recovery on cluster may take more time both because of the communication over net-

work and the data loading overhead from different data node of HDFS.

Since the overhead such as failure detection delay and place rebuilding are relatively in the fixed time. With increasing of input data size, overhead becomes more acceptable. We will further optimize the data recording and recovery in future.

5.2.2 SSCA#1

SSCA#1 is a benchmark of the Scalable Synthetic Compact Applications (SSCA) benchmark suite[2] and it deals with the bioinformatics optimal pattern matching. It matches two sequences by splitting the long one into segments and concurrently comparing the segments with the short one in one place with a single thread. In the test of SSCA#1, we inject a failure after one of the segments matching in a place is just started. Since segments matching contains independent tasks without checkpoint recording, we only inject a fault in the original code without any other manual changes. We simply re-execute the unfinished segment matching in the recovery test.

The results are presented in figure 12 and figure 13. Compared to WordCount, the record-only test in SSCA#1 has a small overhead without recording intermediate data. Hence, even though we record the control flow into HDFS, implement a detection with the Zookeeper, and change the source code to save communication message and trace program states, these parts seem not to have much overhead.

With increasing of data size, the execution time of concurrent tasks increases. Hence, the recovery overhead is hidden while other concurrent tasks are executing. Even though the overhead of detecting failure and rebuilding place increases on cluster, it is still acceptable especially when input size grows.

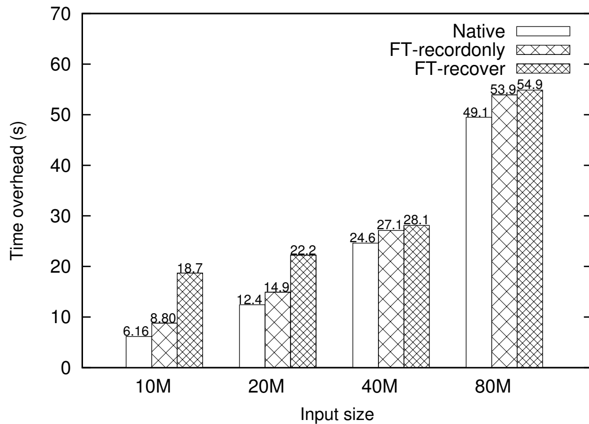

Figure 12: SSCA#1 runs on single machine: 8 places in all, short sequence is 200 characters, and long sequence scales from 10M to 80M

In summary, the record overhead in X10-FT is application-specific. Scientific applications such as SSCA#1 are suitable for our framework than data-parallel applications like WordCount.

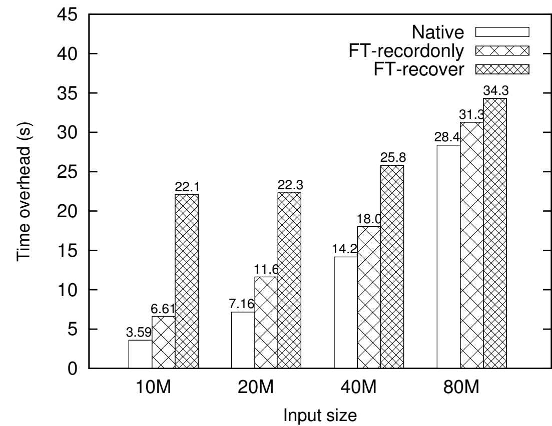

Figure 13: SSCA#1 runs on cluster of 4 machines: 16 places in all, short sequence is 200 characters, and long sequence scales from 10M to 80M

6. RELATED WORK

This section briefly relates X10-FT to prior efforts in fault tolerance, and discuss recent advances in APGAS programming model.

There are a few attempt to providing fault tolerance features to the PGAS model. For example, Ali et al. [1] leverage shadow data structures and redundant remote memory accesses to tolerant potential faults. Vishnu et al. [16] describe a fault resilient, one-sided communication runtime framework, which uses the Global Arrays language and its communication runtime, ARMCI, for the data-centric programming models. In contrast, X10-FT targets the APGAS programming model, which is with more powerful and flexible language features that challenges fault tolerance. Further, X10-FT integrates advances in distributed systems like PAXOS and distributed file systems for more reliable failure detection and recovery.

There are also a number of efforts in providing fault tolerance in other language and runtime, such as CoCheck [14] and FT-MPI [7]. However, even though MPI can also be expressed in X10, the original semantics of X10 language is more powerful than standard MPI, especially for the address space and activity models. Hence, X10-FT directly provides fault tolerance support based on X10 semantics, which essentially add fault tolerance to MPI programs written in X10 as well. Further, virtualization has also been used in HPC [11, 4] 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.

Recently, the X10 language, as a representative of the APGAS model, has been constantly evolving with growing performance [15]. There are also a number of applications, benchmarks and libraries are written in X10 [5, 15, 10]. Hence, adding fault tolerance support would benefit such work with more reliable execution.

7. CONCLUSION AND FUTURE WORK

In this paper, we analyzed the lanaguage features and execution behavior of a typical APGAS language. Based on the analysis, we described the X10-FT framework that extends the APGAS model with fault tolerance support. we proposed a possible design to add fault tolerance features, including the use of DFS, the PAXOS model and some re-

covery solutions based on concepts of *Place* and *Async* of APGAS model.

We described our current implementation of the X10-FT framework, which used the ZooKeeper for fault detection, and then rebuilt the crash places including both launcher process and runtime process. X10-FT also required some modifications to the X10 compiler to insert checkpoints into user code. To durably store checkpoint data, X10-FT integrated HDFS as the persistent storage layer.

We have done a preliminary evaluation for both checkpointing overhead at activity level and a complete functional test of recovering from place failure. Evaluation results showed that X10-FT can successfully recover program execution from node failures, with modest overhead for data-parallel application such as WordCount, and very small overhead for scientific applications. This naturally fit the application domain of X10, indicating the design of X10-FT would be a right direction of adding fault tolerance to X10.

Our future work includes a further implementation of the automatic data dependency calculation and data recovery. We also plan to address issues such as concurrent asynchronous tasks, data consistency, and the failure of main place with a more complete design and validation.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by China National Natural Science Foundation under grant numbered 61003002, a grant from Shanghai Science and Technology Development Funds (No. 12QA1401700), a Foundation for the Author of National Excellent Doctoral Dissertation of PR China and Fundamental Research Funds for the Central Universities in China.

9. REFERENCES

[1] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer. A redundant communication approach to scalable fault tolerance in pgas programming models. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 24–31. IEEE, 2011.

[2] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2:1–10, 2006.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[4] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew. Mercury: Combining performance with dependability using self-virtualization. In *International Conference on Parallel Processing*. IEEE, 2007.

[5] M. Dayarathna, C. Houngkaew, and T. Suzumura. Introducing scalegraph: an x10 library for billion scale graph analytics. In *Proceedings of the ACM SIGPLAN 2012 X10 Workshop*, page 6. ACM, 2012.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] G. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353. Springer Berlin / Heidelberg, 2000.

[8] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[9] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[10] J. Milthorpe, V. Ganesh, A. Rendell, and D. Grove. X10 as a parallel language for scientific computation: practice and experience. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1080–1088. IEEE, 2011.

[11] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32. ACM, 2007.

[12] V. A. Saraswat, B. Bloom, and I. Peshansky. X10 language specification v2.2. http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf, 2012.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[14] G. Stellner. Cocheck: checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pages 526 –531, apr 1996.

[15] O. Tardieu, D. Grove, B. Bloom, D. Cunningham, and B. Herta. X10 for productivity and performance at scale. In *A Submission to the 2012 HPC Class II Challenge*, 2012.

[16] A. Vishnu, H. Van Dam, W. De Jong, P. Balaji, and S. Song. Fault-tolerant communication runtime support for data-centric programming models. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–9. IEEE, 2010.

[17] C. Zhang, C. Xie, Z. Xiao, and H. Chen. Evaluating the performance and scalability of mapreduce applications on x10. *Advanced Parallel Processing Technologies*, pages 46–57, 2011.