Incremental Checkpointing based on Java Source Code Refactoring

(Project Report for CS264 in Spring, 2005)

Thomas Huining Feng
Ptolemy Group, CHESS (Center for Hybrid and Embedded Software Systems)
EECS, UC Berkeley
`http://www.eecs.berkeley.edu/~tfeng/`

Abstract

In this project, incremental checkpointing is developed specifically for Java programs. This checkpointing scheme has a flavor of source code refactoring, which performs almost all the (rule-based) transformation automatically, requiring few (or no in many cases) interaction with the programmer.

Incremental checkpointing bases on a logging technique that records the change in states instead of complete snapshots of states, as is typically done in a serialization scheme. This enables it to be widely applicable to most existing Java programs. Performance loss due to the extra operations for this checkpointing is small enough to make it practical. Real-time property of the checkpointing system may be provable, which will be further explored in our future work.

1 Problem Statement

Checkpointing is widely used in many applications as a fault tolerance technique. [1] As an example of application, many software applications provide the "undo" function, which undoes the recent changes (up to some extent) on the documents that the applications manage. Another example is a robust system that supports fail-recovery. Such a system might maintain a log on its activity. When it crashes unexpectedly, it restarts and recovers its previous state recorded before the crash happened.

1.1 Application Background

The checkpointing solution developed in this project is motivated by the evolution of Ptolemy II [2]. Ptolemy II is a tool for heterogeneous model design and execution.[1] Distributed model execution in Ptolemy II is being implemented. The implementation of a Time Warp [3] style distribution execution requires the checkpointing facility. In such an execution, components belonging to the same model are deployed on multiple machines connected by a network. Each component maintains its local logical time. It is very expensive and even impossible to always make the logical times on those hosts synchronized with the global logical time. As a result, it is normal that there is (possibly big) difference between the the logical times on those hosts. It is then possible for a more advanced hosts to receive a message tagged with a time in its past from another host. In that case, the receiver has to roll back to its previous state, and "replays" all the received messages since that time to reach its new state.

1.2 Requirements

Because of this application background, a very different type of checkpointing is required. It is different from many existing checkpointing strategies in the following ways:

- Memory is considered the stable storage in this scenario, while in many other checkpointing strategies, memory is considered unstable, so checkpointing records must be stored in secondary storage. In this project, hosts themselves are considered stable. They roll back only when inconsistency in the logical times is detected. The rollbacks are requested by the hosts that detect the inconsistency, while in many other strategies, the rollbacks are requested by an external agent, when it sees a hosts crash or do not function properly.
- Many existing checkpointing strategies are in favor of serialization, such as most database systems. There are also strategies that combines serialization and logging, such as those discussed in [4]. However, in this project, we emphasize incrementality. This is because, in this specific application, it is hard to tell what

[1]Ptolemy II is developed by the Ptolemy group at UC Berkeley, headed by Prof. Edward A. Lee. The project is accessable at `http://ptolemy.eecs.berkeley.edu/`.

is a good time to create a serialization snapshot, as each host continuously receives messages and keeps being busy handling those messages. The handling of those messages results in advance in local logical time, which also happens all the time.

- Some existing Ptolemy II models are themselves real-time. Real-time distributed execution of those models is under research. It is desirable to preserve the real-time property of those existing models (to some extent) even after checkpointing support is added.

These special requirements give challenges to the checkpointing that we develope. However, the following consideration leads to a satisfactory solution.

- Because memory storage is considered stable, it is possible to checkpoint the state of object references. This is never possible when we think about crash of programs, which invalids all the memory storage that it uses. Objects residing in memory do not post a problem to our design, while they really do to the others, which rely (completely or partly) on serialization.

- We do not use serialization at any time of this checkpointing. This is for performance reason as well as for checkpointing object references in memory. Performance is an important goal of distributed model execution. We cannot afford the performance loss caused by periodically creating snapshots of the model, which usually consists of tens to thousands of actors.

- Serialization is not suitable for real-time applications, because it is impossible to precisely predict how much work is added by creating a snapshot. Moreover, the extra work to roll back is also unpredictable. David and Willy discussed some checkpointing algorithms featuring the combination of serialization and logging in [4]. It is very often that no snapshot was created exactly at the time to be rolled back. In those cases, the system rolls back to the latest snapshot taken before that time, and replays the messages (recorded in a log) received between the snapshot time and the rollback time. Before an execution, it is not clear how much extra work is done for this purpose. If such an algorithm is implemented in a real-time system, the real-time property is totally lost.

In this project, we develop a checkpointing mechanism completely based on logging.

2 Solution to the Problem based on Java Source Refactoring

Ptolemy II models are built from actors. All the Ptolemy II actors are either written in Java or built by combining other actors. This enables a Java source refactoring approach. In this approach, we focus only on the common nature of all the well-formed Java programs.

A run-time *state* of a piece of Java code, as we are interested in, is the mapping of state variables of an object to their current values. Each *state variable* is defined to be a private non-static field. Its value changes over time. We may define states and state variables formally as follows:

$$S(o,t) = V_{o,t}$$
$$V_{o,t}(offset) = value$$

($S(o,t)$, where o is an object and t is the execution time, evaluates to a function $V_{o,t}$. $V_{o,t}$ is a mapping from offsets of the object to the values stored at those offsets. It is assumed that each offset corresponds to a private non-static field (state variable) of the object. We call $S(o,t)$ a state of the object at that time.)

2.1 Modification on States

With states defined, we can now study different Java features that do modifications on states. In order to roll back at a later time, enough information must be stored to undo those modifications. As modification sites in Java programs are exhaustively studied below, methods to capture them and the information needed to be stored are also made clear.

With a refactoring tool, the modifications are detected in the Java source code. The original source code is then transformed to a new form, in which the recording is done for each modification.

2.1.1 Assignment

The most obvious modification on a state variable at runtime is assignments.

Table 1 shows several examples of transformations on assignments. Most of the transformations discussed in this report (including those shown in Table 1) are based on extra methods. In example 1, the assignment `a = b;` is transformed to a call to an extra method `$ASSIGN$a` with argument `b`. The pseudo-code for this method as as follows:

```
private int $ASSIGN$a(int newValue) {
    ... // Store the old value of a.
    return a = newValue;
}
```

1	`a = b;`
1'	`$ASSIGN$a(b);`
2	`f(a = b);`
2'	`f($ASSIGN$a(b));`
3	`f(..).a = b;`
3'	`f(..).$ASSIGN$a(b);`
4	`f(a = b, g(c = d));`
4'	`f($ASSIGN$a(b), g($ASSIGN$c(d)));`

Table 1: Refactoring Assignments

5	`a += b;`
5'	`$ASSIGN$SPECIAL$a(1, b);`
6	`a -= b;`
6'	`$ASSIGN$SPECIAL$a(2, b);`
7	`a++;`
7'	`$ASSIGN$SPECIAL$a(10, 0);`
8	`++a;`
8'	`$ASSIGN$SPECIAL$a(11, 0);`

Table 2: Refactoring Other Expressions with Side-Effect

This method records the old value of `a` (assuming it is a state variable and its type is `int`), assigns the new value to it, and then return the new value. It exactly models the behavior of an assignment.

As is pointed out, a tool is developed to automatically refactor the Java program. When it detects an assignment to a state variable in the Abstract Syntax Tree (AST) of the Java source code, it converts it into a method call as shown. It also records that the state variable has been assigned at at least one place. Extra fields and extra methods will be added to the class for all the assigned state variables, when the transformer finishes traversing the whole class.

2.1.2 Other Expressions with Side-Effect

Except assignments with the = operator, many other kinds of Java expressions have side-effect. Another extra method is build for all those kinds of expressions, as shown in Table 2.

Extra method `$ASSIGN$SPECIAL$a` handles all the other expressions that have side-effect. Its first argument is the type of the expression. E.g., `1` means +=, `2` means -=, etc. Its second argument is the sub-expression on the right-hand side. In case there is no sub-expression (such as `a++`), the second argument is ignored.

The pseudo-code of this method is similar to the previous one:

```
private int $ASSIGN$SPECIAL$a(int operator,
        int newValue) {
    ... // Store the old value of a.
    switch (operator) {
    case 1:
        return a += newValue;
    case 2:
        return a -= newValue;
    ...
    case 10:
        return a++;
    case 11:
        return ++a;
    ...
    }
    return -1;
}
```

2.1.3 Arrays

Assignments to arrays (or their elements) are handled specially. This is because an array can be modified in different ways. The following piece of code shows such an example:

```
int[][] buffer;
buffer = new int[2][];
buffer[1] = new int[2];
buffer[1][1] = 2;
```

In this example, `buffer` is modified in three different ways (with three different numbers of indices). A different extra method is used in each case.

```
int[][] buffer;
$ASSIGN$buffer(new int[2][]);
$ASSIGN$buffer(1, new int[2]);
$ASSIGN$buffer(1, 1, 2);
```

The extra method used for the first assignment is the same as the one in previous sections. The second method, which takes one more argument as the first index, assigns the new value (`new int[2]`) to the element refered to (`buffer[1]`). Similarly, the third method takes two more arguments as the two indices.

2.1.4 Alias of Arrays

Aliasing is a big problem in C and C++, with which a state variable can be aliased with another name (possibly appearing as a local variable). Modification can then be done on the new name. It is not possible to statically capture the change on those aliased state variables.

In Java, this problem is alleviated. We only consider alias of arrays, because, though objects can be aliased, modification on their states must still be done on private

9	`a = b;`
9'	`a = $BACKUP$b();`

Table 3: Refactoring Array Alias

non-static fields. (Even if `o` is a local variable of an object type, `o.a = b` is still refactored to `o.$ASSIGN$a(b)` if `a` is one of `o`'s state variables.)

When an array is aliased with a local variable, to simplify the solution, its content is backed up in the memory. There is on-going research on alias analysis, but the complexity to precisely determine an alias is sometimes turing-complete (the same as solving the *halting problem*). The examples in Table 3 make it clear how this backup is done with another extra method (assuming that `a` is a local variable, and `b` is a state variable of type `int[][]`). In this example, method call `$BACKUP$b()` backs up the content of `b`, and then returns `b`. In this case `b` is multi-dimensional, so all the elements in all the dimensions are backed up. The following is the pseudo-code of this method:

```
private int[][] $BACKUP$b() {
    for (int i1 = 0; i1 < b.length; i1++) {
        ... // Store the old value of b[i1].
        for (int i2 = 0; i2 < b[i1].length;
                i2++)
            ... // Store the old value of
                // b[i1][i2].
    }
    return b;
}
```

2.2 Checkpoint Management

Each class, once it is refactored, supports checkpointing and rollback operations on its instances.

The notion of checkpoint objects is required to identify the set of run-time objects to be rolled back at a time. A *checkpoint object* is a special object that monitors a set of objects in memory for checkpointing purpose. All the objects monitored by the same checkpoint object must be rolled back together. This simplifies the task of identifying which objects to be rolled back.

2.2.1 Checkpoint Handle

An extra method `$GET$CHECKPOINT` is added to each refactored class. It returns the checkpoint object that monitors the current object. Obviously, calling this method on different objects may yield the same checkpoint object, if those objects are in the same set to be rolled back.

Programmers may directly call methods of a checkpoint object to manage checkpoints. `createCheckpoint` method is used to create a new checkpoint. It takes no argument and returns a checkpoint handle (in the current design, a `long` value). This handle may be used to roll back the system later on.

Contrary to serialization, which takes a system snapshot at the time a checkpoint is created, `createCheckpoint` acturally does nothing but return the next available handle. In this sense, this incremental checkpointing is much more efficient than the traditional serialization scheme.

In our design, the checkpoint handle returned is actually the timestamp denoting the logical execution time on a host. This guarantees that every handle returned is unique. Programmers can thus use those timestamps to specify the exact time to be rolled back. However, this design may be changed in the future. In reality, any unique handle might be returned. Programmers should not assume any meaning in those handles.

2.2.2 Storing the Old Values

When we discussed the extra methods above, we ignored the parts where old values of the state variables are stored. Those parts are more easily understood when the notions of checkpoint objects and handles are made clear.

To store an old value, the methods first checks whether a checkpoint handle has been returned by the `createCheckpoint` method of the checkpoint object that monitors the current object. If not, nothing needs to be done, since the program cannot roll back to a previous state without a handle. Otherwise, the current value of the state variable is pushed to a state record, which is of a stack structure.

When an old value is pushed to the state record, the current timestamp of the checkpoint object is associated with it. This timestamp is used to judge whether to restore the old value when the program tries to roll back to a previous time.

2.2.3 Joining a Set

Checkpoint sets, each monitored by a checkpoint object, is not static at run-time. In stead, objects frequently join and leave those sets. This is an effect of object assignment.

Let us reconsider the simple assignment `a = b` where `a` is a state variable of the current object (`this` object). `a` has an object type, which is the same as `b`. As discussed above, this assignment is refactored to `$ASSIGN$a(b)`, which has the same effect as the assignment itself. However, if both `this` object and `b` are instances of classes that have been refactored, they should be placed in the same checkpoint set, if they are not already in the same set. This makes it possible to roll back both objects at the same time later. The following piece of code further shows why this is necessary:

```
a = b;
// Create a checkpoint.
int handle =
    $GET$CHECKPOINT$().createCheckpoint();
b.i = 1;
// Rollback.
$GET$CHECKPOINT$().rollback(handle);
```

After refactoring, the above piece of code becomes:

```
$ASSIGN$a(b);
// Create a checkpoint. Not refactored.
int handle =
    $GET$CHECKPOINT$().createCheckpoint();
b.$ASSIGN$i(1);
// Rollback.
$GET$CHECKPOINT$().rollback(handle);
```

If `this` object and `b` were not places in the same set after the first assignment, rolling back with the handle would be useless, because the handle was returned by the checkpoint object monitoring `this` object, not `b`. It is possible to roll back the effect on an object monitored by another checkpoint object.

With this consideration, an extra method `$SET$CHECKPOINT` is added to set the checkpoint object of an object. This method is called in the `$ASSIGN$a` method if `b`'s class has been refactored.

An object may also join a set at a field declaration. The following is such an example:

```
class A {
    ...
    private B f = new B();
    ...
}
```

In this example, a new instance of `B` is created and immediately assigned to the `f` field of an `A` object. According to our design, both objects must be in the same checkpoint set, so that later when we roll back the `A` object, the content of its `f` field is also rolled back. Hence, the transformer gives the following refactoring result:

```
class A {
    ...
    private B f =
        new B().$SET$CHECKPOINT($CHECKPOINT);
    ...
    // Here is the checkpoint object that
    // monitors the current object.
    CheckpointObject $CHECKPOINT = ...;
}
```

To make this refactoring legal, extra method `$SET$CHECKPOINT` is deliberately designed to return `this`.

2.3 Cross Analysis

In many cases we need to cross-analyze multiple classes. Suppose we are currently analyzing class `A`, which refers to class `B`. Whether class `B` is refactored affects the refactoring result of `A`.

Even though we only refactor private fields of one class at a time, it is still possible for another class that is not being analyzed to directly write to those fields. For example, enclosed classes can always access the private fields of their enclosing classes. To deal with this, modification in those enclosed classes must also be refactored in a similar way.

The `$SET$CHECKPOINT` method is also affected by cross-analysis of other classes. When this method is called, the checkpoint object of the current object is changed. To maintain the consistency of checkpoint sets, all the other objects that it refers to must also change their checkpoint object. The pseudo-code of this method in class `A` is shown below:

```
class A {
    private B b;
    ...
    CheckpointObject $CHECKPOINT = ...
    public A $SET$CHECKPOINT(Checkpoint cp) {
        if ($CHECKPOINT != cp) {
            $CHECKPOINT = cp;
            b.$SET$CHECKPOINT(cp); // (*)
        }
        return this;
    }
}
```

Statement `(*)` is there only if class `B` is also refactored. We do not require programmers to refactor every class. In fact, it is not possible to refactor all the classes used, because many classes are in binary form in the Java built-in library or some other third-party libraries.

2.4 Class Substitution

Some Java programs store states in special built-in Java objects, such as hash tables and linked lists. To roll back, it is also necessary to roll back those built-in objects. This posts a question to our design. The Java built-in library is not modifiable. Even if it is, it is not a good idea to modify the library and distribute it to every end-user of the refactored code.

Our solution is given by refactoring a re-implementation of part of the built-in library. This re-implementation, though it could be written manually following the Java API, is retrieved from GCJ[2], a GNU compiler for Java. The source code of some built-in classes are downloaded from GCJ, and the same refactoring is applied to the source code to get new classes that supports checkpointing. The refactored classes are placed in a new package. After that, when the tool refactors a class using one of those built-in classes, a simple renaming is done so that the class uses the refactored GCJ classes.

For example, such classes as `java.util.Hashtable` and `java.util.LinkedList` are retrieved from the GCJ CVS repository. The refactored classes are `ptolemy.backtrack.java.util.Hashtable` and `ptolemy.backtrack.java.util.LinkedList`. When `java.util.Hashtable` is seen in a user class to be refactored, `ptolemy.backtrack.java.util.Hashtable` is used instead, which supports checkpointing.

3 Evaluation

The refactoring tool has been implemented in Java. It makes use of Eclipse[3] JDT's AST builder. This refactoring mechanism has been successfully applied to over 50 Ptolemy II actors and over 20 GCJ classes in its `java.util` package. This shows that this refactoring mechanism is widely applicable to most Ptolemy actors as well as many general-purpose Java programs.

Performance evaluation of this tool is carried out in various aspects.

3.1 Performance of Refactoring

The refactoring tool is designed in such a way that the program AST is traversed only once. In this one traversal, it type-checks the Java program, and also refactors the program by calling different handlers. (Those handlers are defined by refactoring rules.) The time for this tool to refactor a program is about the same as the time for Javac (the Java compiler) to compile it.

[2]GCJ homepage: `http://gcc.gnu.org/java/`
[3]Eclipse homepage: `http://eclipse.org/`

3.2 Run-Time Performance of the Refactored Code

Obviously, the run-time performance of the refactored code heavily depends on how many state variables there are in the program, and how often they are modified. The more often state variables are modified, the more extra work has to be done in order to back up their old value, and also the more memory space is used to store the change history.

We are not considering aliasing in this performance evaluation. Frequently aliasing an array may cause extremely expensive array cloning. The programmers should be aware of this and avoid aliasing as much as possible.

Assignments in the original program become method calls after refactoring. In the body of those methods, extra work is done to store the old values. This accounts for a *constant* amount of extra work for each modification to a state variable. In a worst-case program that modifies state variables in each statement, a *linear* degradation in performance is seen.

However, the programmers are assumed to be aware of this refactoring, though they typically need not create their program in a special way. With this awareness, the programmers should not modify the state variables heavily. Specifically, they should not use state variables as loop variables, which are modified in each iteration of a loop. Frequent modifications should be done on local variables instead. Only when the values of the local variables are relatively stable, e.g., at the end of method bodies, are they finally assigned to state variables. If this is the case, the performance loss due to checkpointing is neglectable.

3.3 Real-Time Property

As pointed out above, real-time property in some Ptolemy models is important. With this checkpointing strategy, the refactored models, which support rollback, may preserve their real-time property. This is because of the linear performance degradation, which can be precisely predicted before an execution. This aspect is being studied in our group.

References

[1] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra M. R. Kintala. Checkpointing and its applications. In *Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.

[2] Edward A. Lee. Overview of the ptolemy project. Technical report, Technical Memorandum UCB/ERL M03/25, July 2003.

[3] David R. Jefferson. Virtual time. *ACM Transaction on Programming Languages and Systems*, 7(3):404–425, 1985.

[4] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.