Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

INCREMENTAL CHECKPOINTING WITH APPLICATION TO DISTRIBUTED DISCRETE EVENT SIMULATION

Thomas Huining Feng
Edward A. Lee

Center for Hybrid and Embedded Software Systems (CHESS)
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, U.S.A.

ABSTRACT

Checkpointing is widely used in robust fault-tolerant applications. We present an efficient incremental checkpointing mechanism. It requires to record only the state changes and not the complete state. After the creation of a checkpoint, state changes are logged incrementally as records in memory, with which an application can spontaneously roll back later. This incrementalism allows us to implement checkpointing with high performance. Only small constant time is required for checkpoint creation and state recording. Rollback requires linear time in the number of recorded state changes, which is bounded by the number of state variables times the number of checkpoints. We implement a Java source transformer that automatically converts an existing application into a behavior-preserving one with checkpointing functionality. This transformation is application-independent and application-transparent. A wide range of applications can benefit from this technique. Currently, it has been used for distributed discrete event simulation using the Time Warp technique.

1 INTRODUCTION

Checkpointing is a recovery technique widely used in robust fault-tolerant applications. For example, many contemporary database applications have built-in recovery mechanisms, with which they can recover data from unintended destructive operations, storage failure, or program crash. Safety-critical applications also employ recovery mechanisms to ensure that once unexpected situations occur, they can still restore sensible states and continue to function correctly. These applications create checkpoints during normal execution. Those checkpoints record the information necessary to recover the states in case of certain types of failure. The types of tolerable failure are very application-dependent. For example, document editing applications may attempt to tolerate software bugs, but they generally assume protected memory segments and disks to be reliable storage; database applications may tolerate some disk failure by using Redundant Array of Independent/Inexpensive Disks (RAID) or system backups; safety-critical applications do not rely on any single type of storage, but make use of various kinds of storage devices to ensure maximum fault-tolerance capability.

We develop an incremental checkpointing mechanism here. It does not require to take snapshots of the complete execution state. The applications can execute asynchronously with their checkpointing systems. The application states are changed during the execution. Every independent change is recorded in the most recent checkpoint at a small constant cost. Later, the applications can spontaneously restore their state by rolling back to the checkpoints.

We have employed this incremental checkpointing mechanism in our modeling and simulation environment, Ptolemy II (Brooks et al. 2005). Instead of restricting the mechanism to be applicable for Ptolemy II only, we view this use case as a concrete example of a rich set of potential applications. Therefore, *application-independence* is an important property that we try to pursue. Besides this, our implementation is also *application-transparent* so that application designers need not consider low-level checkpointing details. To achieve these goals, we take the program refactoring (Fowler 1999) approach, and invent a source to source transformer. It accepts the source of existing applications, and outputs behavior-preserving applications with extra functions for checkpointing. Very little human interaction is required in this process.

The rest of the paper is organized as follows: Section 2 offers an overview of our checkpointing strategy. In Section 3, the source transformation is discussed. In Section 4, operations for checkpoint management are provided so that the applications can spontaneously interact with their checkpointing systems at run time. Our simulation environ-

ment is presented in Section 5 as an application. Related work is studied in Section 6. Section 7 concludes this work.

2 OVERVIEW OF THE CHECKPOINTING STRATEGY

Application-independence and application-transparency (Strom and Yemini 1985) are two important goals of our checkpointing strategy:

- Application-independence: We make no assumption on the nature of the target applications. The analysis and transformation method is generally applicable to the source of many existing applications. Currently, we have an implementation for arbitrary Java programs, but other languages can also be supported using similar techniques.
- Application-transparency: We try to free application designers from considering low-level checkpointing details. In some other checkpointing approaches (Lawall and Muller 2000), the designers are required to manually construct their applications in a special way for checkpointing. This is not required in our approach. A program analyzer is implemented to automatically extract state information. With this information, behavior-preserving applications can be generated with the program transformer.

2.1 The Problem

State recovery is commonly required in applications. However, there is no uniform definition of application state. General-purpose applications usually consider the contents of their accessible memory as their state, because this memory contains the objects that they operate on. For example, simulation environments store the run-time model state in memory, and document editing applications store documents in memory when they are edited by the users. For these applications, it is sufficient to record the history of memory writes in the checkpoints.

Our research aims to develop an efficient checkpointing mechanism for the above-described applications. We do not try to handle the state of external devices. For checkpointing of these states, we fall back to the traditional approach by requiring the programmers to provide extra methods.

We further assume that the affected applications themselves issue checkpoint and rollback requests. Our simulation environment is one such example. It simulates distributed discrete event models using Time Warp (Jefferson 1985). It requests to roll back its own state when a causality conflict is detected. Document editing applications can be another example. They allow users to undo some editing operations. Under this assumption of spontaneity, we do not address the problem caused by arbitrary unexpected crash that completely invalidates the running application.

2.2 A Program Analysis and Transformation Approach

The state needs to be discovered before a mechanism can be provided to record it. Because we define the state to be the contents of application-accessible memory, we can use a program analyzer that statically analyzes memory access in the source code. For applications written in object-oriented languages such as Java, the state is accessed by means of object fields. We currently only consider private fields as application state. This assumption is not necessary for the correctness of this technique, but it allows us to simplify the analyzer design. Because private fields can only be modified in the Java files that define them, the analyzer can precisely detect all the modification sites in those files. (Public fields and protected fields can also be supported, either by extending the analyzer to analyze all the Java files, or by transforming them into private fields with get/set methods generated.)

At run time, the checkpointing system maintains the application state once a checkpoint is created. We consider the checkpointing system's private memory as a stable storage invisible from the application itself. State changes are logged in that memory.

Extra code is required to log the changes. This code is scattered throughout the application. It is hard and error-prone for the programmers themselves to write this code by hand. Therefore, we develop a program transformer that exploits the information from the analyzer, and automatically inserts this extra code at the program points where state is change.

3 SOURCE TRANSFORMATION

In this section, a program transformation method is presented for Java. It automatically inserts checkpointing code at the program points where state can be changed at run time. This is essentially an aspect-oriented programming (Kiczales et al. 1997) approach, as we define the aspects and also provide a tool to weave those aspects with the target programs.

Though we currently assume that the applications are written in Java, our method is generally applicable to other languages with some assumptions. Specifically, we assume no pointer aliasing, no pointer arithmetic, and automatic memory management (available in some libraries such as Boehm's garbage collector — see Boehm and Demers 1997). These assumptions may be met by some C++ applications.

3.1 Analysis

Our method starts with an analysis phase. The analyzer performs an intra-procedural analysis on all the Java classes

that need checkpointing. It extracts the following information:

1. All the private fields of those classes and their types.
2. All the operations in the code that can modify the private fields.
3. The class hierarchy. (Not all private fields are explicitly defined in the classes. Some of them may be implicitly inherited from superclasses.)

There are commonalities between this analyzer and the a Java compiler. In a Java compiler, information types 1 and 3 are obtained from the type checker, while type 2 is examined by the scoping and visibility checker.

3.2 Assignment Transformation

Assignments may modify program state. Intuitive examples of assignment transformations are given in Table 1. The assignments in the original source are transformed to calls of auxiliary methods. In Example 1, where `a` is a private field of type `int`, the assignment `a = b` becomes a call to `$ASSIGN$a` with argument `b`. `$ASSIGN$a` is automatically generated for the current class:

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

This method records the old value of `a`, assigns the new value to it, and then returns the new value. It precisely models the observable effect of an assignment expression in Java.

For object assignments, the transformer generates the same auxiliary methods. This means that only object pointers are stored for the checkpoints. (Object assignments in Java are essentially pointer assignments.) Cloning or deep copy is not necessary. Therefore, the cost for logging an assignment is always a small constant, no matter what type the field has.

In operational semantics, the following rule formally defines Java assignments:

$$\frac{< e, \sigma > \Downarrow n}{< x = e, \sigma > \Downarrow \sigma[x := n]} \tag{1}$$

According to this rule, if expression e is evaluated to number n in the abstract program state σ, then the new state after executing $x = e$ is the same as σ, except that the value of variable x becomes n (denoted by $\sigma[x := n]$).

In the transformed code, we can imagine that a checkpoint is used to record the change history. We may use

Table 1: Examples of Assignment Transformations.

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)));`

φ to denote the *current checkpoint*. (The application may create a sequence of checkpoints at run time, and the latest one among them is current.) We can now define new rules that correspond to the auxiliary methods for assignments:

$$\frac{< e, (\sigma, \varphi) > \Downarrow n \quad \sigma(x) == n_0 \quad \varphi(x) == undefined}{< x = e, (\sigma, \varphi) > \Downarrow (\sigma[x := n], \varphi[x := n_0])} \tag{2}$$

$$\frac{< e, (\sigma, \varphi) > \Downarrow n \quad \varphi(x) == n_0}{< x = e, (\sigma, \varphi) > \Downarrow (\sigma[x := n], \varphi)}. \tag{3}$$

We extend the program state from σ to tuple (σ, φ). In Rule (2), we define that if all the following conditions are satisfied, then the new state after the assignment is the same as (σ, φ), except that $\sigma(x)$ becomes n (the new value), and $\varphi(x)$ becomes n_0 (the old value):

1. Expression e evaluates to n in state (σ, φ);
2. The old value of x is n_0; and
3. x does not have an old value recorded in φ.

On the contrary, if an old value of x has already been recorded in φ, Rule (3) applies. In this case, σ updates, but φ remains the same. Rule (3) is defined mainly for efficiency (both time and space). For each checkpoint, there is only one execution point ep to which the program can roll back, so there is no need to store the old value more than once. To retain multiple execution points $\{ep_1, ep_2, \ldots, ep_n\}$ for rollback, a sequence of checkpoints can be created, and the states are recorded in them incrementally.

3.3 Special Expressions with Side-Effects

Some Java expressions have side-effects similar to assignments. For example, operators such as "+=" and "++" update their operands. When they are found in the program, auxiliary methods are created to simulate them after recording the old values. Their operational semantics can be defined similarly.

3.4 The Data Structure

The checkpointing system uses multiple stacks to keep track of the change history of program states. Figure 1 shows an example with two private fields. A different stack is allocated for each. At run time, assignments update these fields sequentially. Asynchronously, the program also creates a sequence of checkpoints. In this example, `Assign1` is an assignment to `field1`. No record is kept for this assignment simply because no checkpoint exists at the beginning. When `Assign2` modifies `field2`, a record is kept in `field2`'s stack `record2`. This is necessary, because `checkpoint1` is created before this, and a later rollback requires `field2`'s old value. The current checkpoint is associated with this record in the stack. Later, rollback with this checkpoint or earlier checkpoints will use this record.

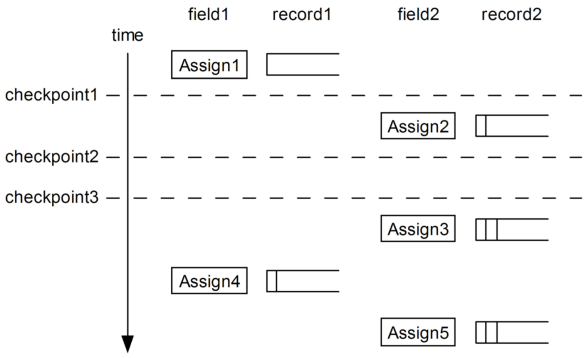

Figure 1: Using Two Stacks to Record the Change History of Two Private Fields.

For `checkpoint2`, however, no record needs to be added, because `checkpoint2` and `checkpoint3` are semantically equivalent. `Assign3` and `Assign5` are two more assignments to `field2`. A record is kept for `Assign3` but not for `Assign5`, because none of the checkpoints will require `field2`'s value before `Assign5`.

At the end of this example, if the program rolls back to `checkpoint2` or `checkpoint3`, the effect of `Assign3` and `Assign4` has to be canceled with the records in the stacks. If it rolls back to `checkpoint1` instead, `Assign2` and `Assign4` need to be undone, but the record for `Assign3` will simply be discarded.

Conceptually, a different stack is allocated with every private field (except arrays, discussed below). Globally, the checkpointing system also uses a stack to store all the checkpoints that have been created. For space efficiency, we do not allocate stacks for provably unmodified fields.

3.5 Arrays

Array assignments may require special handling, because an array can be modified in different ways as shown in this example:

```
int[][] buffer;
...
buffer = new int[2][];
buffer[0] = new int[2];
buffer[0][1] = 2;
```

Here, `buffer` is assigned to with 0, 1, or 2 indices. A different auxiliary method is needed for each case:

```
int[][] buffer;
...
$ASSIGN$buffer(new int[2][]);
$ASSIGN$buffer(0, new int[2]);
$ASSIGN$buffer(0, 1, 2);
```

The first auxiliary method has type signature "`int[][] $ASSIGN$buffer(int[][])`", and is the same as the one introduced before. The second one, which takes one more argument as the array's first index, assigns a new value (of type `int[]`) to the element referred to. The third one takes two index arguments. These auxiliary methods use different stacks to record the old values. The indices are also recorded, so that the changes can be undone for the affected elements only.

Array aliasing is also problematic. An array field can be aliased with another name, possibly appearing as a local variable or as a formal parameter to a method. (Objects can also be aliased. However, to directly modify an aliased object, the Java program still needs to explicitly access its fields. This can be captured without specialty. For example, `o.a = b` will be transformed to `o.$ASSIGN$a(b)` if `o.a` is private, regardless of whether `o` is a local variable.) In our approach, before an array is aliased, its contents are backed up in the memory with another auxiliary method. It performs a possibly multi-dimensional copy for the array. This copy is linear in the array size. In practice, usually only a small part of the array will actually change after the aliasing. A full copy may not be necessary. On-going research on alias analysis helps to predict the changed part of an aliased array (Diwan, McKinley, and Moss 1998). However, the complexity of a precise analysis may be unacceptable. Therefore, we do not include alias analysis in our current implementation.

3.6 Class Substitution

Applications may also store their states in native Java objects, such as hash tables and linked lists. These hidden states also need to be recovered.

We decide not to modify the existing Java library. Instead, we obtain part of its source code, and apply the same transformation to it. The generated classes are packaged specially for checkpointing. When the transformer detects instantiations of the state-keeping classes in the Java library, it substitutes them with the classes in the checkpointing package.

Random number generators are useful in applications such as simulators for probabilistic models. Note that `Random` is also a native Java class with a state, which is the current random seed. With its transformed version, the checkpointing system is able to roll back the random seed. The same sequence of random numbers will be generated after the rollback. Simulators may exploit this property, and reproduce the probabilistic simulations.

3.7 Soundness under Assumptions

We argue that our checkpointing approach is sound under some assumptions. By *soundness* we mean that no observable difference remains in the program states after a rollback. The internal state of the Java Virtual Machine (JVM) may become different, but we ignore this difference as long as it cannot be observed by the program itself. Note that the internal state of the checkpointing system itself is not observable from the program, either.

Our assumptions are:

- States are only stored in private non-static fields. (It is dangerous to roll back static fields in a multi-threaded environment.)
- If states in external devices (e.g., hard disks, network and human interaction devices) need to be rolled back, extra methods are provided by the programmers to handle them.
- If state-keeping classes in libraries (such as the Java standard library) need to be rolled back, they are transformed, and class substitution is performed in the application source code.
- All checkpointing operations, including checkpoint creation, state recording, and rollback, are performed atomically.

A proof of soundness can be obtained by a thorough study of all the Java language features that a program may use to change its states or observe the changes.

4 CHECKPOINT MANAGEMENT

During execution, the applications can create checkpoints or roll back to previously created checkpoints. This is achieved by directly invoking methods in the checkpointing system.

4.1 Checkpoint Creation

The transformer adds method "`CheckpointObject $GET$CHECKPOINT()`" to each transformed class. This method returns the *checkpoint object* for any instance of that class. A checkpoint object monitors a set of Java objects at run time. Its method "`long createCheckpoint()`" is used to create checkpoints for those Java objects. This method increases the global checkpointing timestamp (an increasing static `long` value). The new timestamp is returned as a *checkpoint handle*, a globally unique identifier for the newly created checkpoint. This checkpointing operation takes only small constant time.

The checkpoint objects monitor disjoint sets of Java objects. In our implementation, we define these sets of Java objects to be the basic unit of checkpointing and rollback operations.

4.2 Unification of Checkpoint Objects

At run time, checkpoint objects monitor changing sets of Java objects. Two sets may be unified so that only one checkpoint object remains to monitor the new set. A motivating example for this situation is given below. In this example, `a` is a private object field of the current class; `b` is another object of a compatible type.

```
a = b;
// Create a checkpoint.
long handle =
  $GET$CHECKPOINT$().createCheckpoint();
b.i = 1;
// Roll back.
$GET$CHECKPOINT$().rollback(handle);
```

After the transformation, the above piece of code becomes:

```
$ASSIGN$a(b);
// Create a checkpoint.
long handle =
  $GET$CHECKPOINT$().createCheckpoint();
b.$ASSIGN$i(1);
// Roll back.
$GET$CHECKPOINT$().rollback(handle);
```

Assume that `this` object (the object on which the method is invoked) and `b` are initially monitored by two

different checkpoint objects. If no extra care is taken, after the rollback, the state of `b` will not be restored because it belongs to another set that does not contain `this` object. As a consequence, the change of `b.i` is still observable from `this` object with `a.i`. The naive solution of simply changing `b`'s checkpoint object when it is assigned to `a` will not work in general, because the objects that `b` refers to may still contain observable changes. A correct solution requires that the auxiliary method `$ASSIGN$a` unify the two sets of Java objects, and form a new checkpoint object that monitors the union set. Therefore, the rollback operation in this example is actually called on the unified checkpoint object.

The use of multiple checkpoint objects allows the program to record and roll back only part of its state. For example, a simulator may decide to roll back the memory contents corresponding to the running model's state, but keep the state changes in other components such as user interface and debugger. In this case, it will only roll back the checkpoint object that monitors the model's simulation state.

4.3 Rollback and Discard

As discussed above, the rollback operation affects only the set monitored by the checkpoint object. The changed private fields are traversed in a depth-first search (DFS). Their old values are restored with the values in their stacks. After rollback, the used checkpoint and other newer ones are discarded. The memory allocated for the records will be reclaimed by the Java garbage collector.

The discard operation is similar to rollback, except that it only discards the records without restoring the values. The memory will also be reclaimed.

Rollback and discard, unlike other checkpointing operations that take constant time, have linear complexity in the number of changes recorded after the checkpoint creation time. Therefore, applications aiming for high performance should not invoke these operations frequently.

5 APPLICATION: A SIMULATION FRAMEWORK FOR EMBEDDED SYSTEMS

Our checkpointing technique has many applications. The simulation environment, Ptolemy II, developed at EECS, UC Berkeley, is an example of an application. It is a Java-based framework for model-based design and simulation of embedded systems. The need for a dynamic state recovery mechanism arises as we develop distributed discrete event simulation using Time Warp (Jefferson 1985). In our system, distributed collaborating components in the model keep track of their local virtual times, with which their event handlers decide whether events are imminent. The system also keeps track of the Global Virtual Time (GVT), a lower bound of the local times. We allow the local times to differ from each other. The faster components do computations in advance, optimistically assuming this is safe. This type of simulation may yield a significant performance improvement (compared to traditional distributed simulations where components advance time synchronously). However, as a consequence, the faster components may receive events sent in their past from slower ones, giving rise to causality conflicts. To maintain global consistency, on receiving past events, the components must recover their previous state by undoing the optimistic computations. (These components should also cancel the messages sent as the outcome of the computations, but here we do not address a specific mechanism to achieve this. The reader is referred to Das et al. 1994.)

We take the program transformation approach to provide state recovery for the simulator. In our case, the programs are in fact distributed models constructed by connecting basic building blocks written in Java. The transformation tool takes any existing model, and converts it into a new one with checkpointing functionality. In a simulation, the new model may create a checkpoint every time its components process events or advance their local times. When a past event is received, the affected component rolls back with a previously created checkpoint. This rollback precisely sets back its local time to the event time, which is always greater than or equal to GVT. This time-advancing guarantee helps to avoid Domino Effect (Strom and Yemini 1985).

Whenever the whole system advances the GVT, the components discard the older checkpoints to reclaim memory.

6 RELATED WORK

Software recovery or fault-tolerance is being actively studied by a number of researchers. Serialization is a straightforward method. Many contemporary languages, such as Java and C#, provide built-in serialization mechanisms. However, these require the programmers to explicitly define the methods to record the states into output streams and conversely, to restore the states from input streams. Some of the drawbacks are listed below:

- It places extra burden on the programmers by requiring them to implement the serialization and deserialization methods;
- It is not efficient enough, because the contents of objects and arrays, instead of their references, are stored in the streams.
- It is difficult to determine what portion of the state will change in the future. To be exhaustive, programmers usually take a conservative approach by serializing the entire state.

Unlike serialization, incremental backup has been implemented in many database systems such as Berkeley DB (Olson, Bostic, and Seltzer 1999), Oracle (Greenwald, Stackowiak, and Stern 2001) and MySQL. It is an efficient backup technique that makes it possible to restore external data after a system crash. Because it has the assumption that data are stored externally, it does not solve the problem of discovering and recording the application state in memory.

Software transactional memory (STM) (Shavit and Touitou 1995) is another related technique. It provides transactions which guarantee atomic reads and writes to shared memory. In a parallel system, before a process can write to a shared memory block, it starts a transaction. In the transaction, the write is performed on a separate copy invisible from other processes. When the write is finished, a *test-and-commit* operation is issued. It commits the write only if the shared memory block has not been modified concurrently by other processes. We believe that this approach is in a sense orthogonal to ours. It guarantees atomicity, but a committed change cannot be rolled back. In our approach, atomicity is not guaranteed, but the state can always roll back (provided that there is enough memory for the records).

There has been research on incremental state saving for C++ and executable code. For example, in Steinman (1993) an efficient state saving mechanism is provided for C++. However, it cannot be used for Java without change. Moreover, reference manipulations are not guaranteed to be rollbackable. Dynamic data structures, such as lists and trees, need special treatment. C++ templates are used in Bruce (1995) and Rönngren, Liljenstam, Ayani, and Montagnat (1996) to capture state changes in the source. However, many other languages do not support templates, and heavy use of templates significantly increases compile time and code size. In West and Panesar (1996), executable code is directly analyzed and modified to capture state changes. However, due to the loss of source-level information, it is not easy to detect states precisely. The vase variaty of executable formats also limits its application.

A Java-based incremental checkpointing mechanism similar to ours is independently proposed in Lawall and Muller (2000). It can be considered as incremental serialization for Java. Programmers need to provide extra methods to incrementally record states in output streams. Compared to their approach, our incremental checkpointing mechanism has these advantages:

- No extra methods need to be provided by programmers. The transformer automatically generates the auxiliary methods.
- States are automatically determined by the program analyzer. Programmers need not identify the fields that will change in the future.
- We do not record state in streams. Instead, we store only object fields' old references in memory when they are assigned to. As a result, every state change incurs only a constant overhead.

7 CONCLUSION

Many applications, including our distributed simulation environment with Time Warp, require run-time state recovery. An incremental checkpointing mechanism is developed here. This mechanism is based on program analysis and transformation. We automate this process with a tool that takes Java source code as input, and outputs new code that supports incremental checkpointing. The transformed applications can create checkpoints and recover their states dynamically. Programmers are thus freed from dealing with checkpointing details. Frequently performed operations, such as checkpoint creation and state recording, take only small constant time. Rollback and discard are the less frequently performed operations that take linear time in the number of recorded state changes, which is bounded by the number of state variables times the number of checkpoints.

ACKNOWLEDGMENTS

This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and CHESS (the Center for Hybrid and Embedded Software Systems), which receives support from NSF and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

REFERENCES

Boehm, H.-J., and A. J. Demers. 1997. A garbage collector for C and C++. <http://www.hpl.hp.com/personal/Hans_Boehm/gc/>.

Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. 2005. Ptolemy II - heterogeneous concurrent modeling and design in Java. Technical Report UCB/ERL M05/21, EECS, UC Berkeley.

Bruce, D. 1995. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 40–49: IEEE Computer Society.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. GTW: a time warp system for shared memory multiprocessors. In *Proceedings of the 26th Winter Simulation Conference*, 1332–1339.

Diwan, A., K. S. McKinley, and J. E. B. Moss. 1998. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming*

Language Design and Implementation, 106–117. New York, NY, USA: ACM Press.

Fowler, M. 1999. *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Greenwald, R., R. Stackowiak, and J. Stern. 2001, June. *Oracle essentials: Oracle9 i, Oracle8 i and Oracle8*. 2nd edition. O'Reilly & Associates, Inc.

Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Language and Systems* 7 (3): 404–425.

Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, ed. M. Akşit and S. Matsuoka, Volume 1241, 220–242. Berlin, Heidelberg, and New York: Springer-Verlag.

Lawall, J. L., and G. Muller. 2000. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, 61–70. New York, NY, USA: IEEE.

Olson, M. A., K. Bostic, and M. I. Seltzer. 1999. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, 183–191.

Rönngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. 1996. Transparent incremental state saving in time warp parallel discrete event simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, 70–77.

Shavit, N., and D. Touitou. 1995. Software transactional memory. In *Symposium on Principles of Distributed Computing*, 204–213.

Steinman, J. S. 1993. Incremental state saving in SPEEDES using C++. In *Proceedings of the Winter Simulation Conference*, 687–696.

Strom, R., and S. Yemini. 1985. Optimistic recovery in distributed systems. *ACM Transactions on Programming Language and Systems* 3 (3): 204–226.

West, D., and K. Panesar. 1996. Automatic incremental state saving. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, 78–85.

AUTHOR BIOGRAPHIES

THOMAS HUINING FENG is a Ph.D. student at the Electrical Engineering and Computer Sciences (EECS) department at U.C. Berkeley. He is a member of the Berkeley Ptolemy project, headed by Prof. Edward A. Lee. His research interests include heterogeneous modeling and simulation, distributed systems and Time Warp simulation, automatic program analysis and transformation, software fault tolerance, and software engineering. He obtained his Bachelor's degree from Nanjing University in China, and his Master's degree (M.Sc) from McGill University in Canada. His e-mail address is <tfeng@eecs.berkeley.edu>, and his web page is <http://ptolemy.eecs.berkeley.edu/~tfeng/>.

EDWARD A. LEE is a Professor and Chair of the Electrical Engineering and Computer Sciences (EECS) department at U.C. Berkeley. His research interests center on design, modeling, and simulation of embedded, real-time computational systems. He is a director of Chess, the Berkeley Center for Hybrid and Embedded Software Systems, and is the director of the Berkeley Ptolemy project. He is co-author of five books and numerous papers. He has led the development of several influential open-source software packages, including Ptolemy, Ptolemy II, HyVisual, and VisualSense. His bachelors degree (B.S.) is from Yale University (1979), his masters (S.M.) from MIT (1981), and his Ph.D. from U.C. Berkeley (1986). From 1979 to 1982 he was a member of technical staff at Bell Telephone Laboratories in Holmdel, New Jersey, in the Advanced Data Communications Laboratory. He is a co-founder of BDTI, Inc., where he is currently a Senior Technical Advisor, and has consulted for a number of other companies. He is a Fellow of the IEEE, was an NSF Presidential Young Investigator, and won the 1997 Frederick Emmons Terman Award for Engineering Education. His e-mail address is <eal@eecs.berkeley.edu>, and his web page is <http://ptolemy.eecs.berkeley.edu/~eal/>.