Efficient Incremental Checkpointing of Java Programs*

Julia L. Lawall† and Gilles Muller
COMPOSE group, http://www.irisa.fr/compose
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jll,muller}@irisa.fr
tel:+33.2.99.84.72.87, fax:+33.2.99.84.71.71

Abstract

This paper investigates the optimization of language-level checkpointing of Java programs. First, we describe how to systematically associate incremental checkpoints with Java classes. While being safe, the genericness of this solution induces substantial execution overhead. Second, to solve the dilemma of genericness versus performance, we use automatic program specialization to transform the generic checkpointing methods into highly optimized ones. Specialization exploits two kinds of information: (i) structural properties about the program classes, (ii) knowledge of unmodified data structures in specific program phases. The latter information allows us to generate phase-specific checkpointing methods. We evaluate our approach on two benchmarks, a realistic application which consists of a program analysis engine, and a synthetic program which can serve as a metric. Specialization gives a speedup proportional to the complexity of the object structure and the modification pattern. Measured speedups for the program analysis engine are up to 1.5, and for the synthetic program are up to 15.

1 Introduction

Checkpointing is known to introduce overhead proportional to the checkpoint size [12, 28]. Traditionally, optimizations of the checkpointing process are targeted toward scientific programs written in Fortran or C. Such programs often have good locality and large regions of read-only data. In this environment, an effective optimization technique is *incremental checkpointing*, which uses system-level facilities to identify modified virtual-memory pages [7, 19, 25]. Each checkpoint contains only the pages that have been modified since the previous checkpoint. Additionally, by using a mechanism such as copy-on-write, the application need not be blocked, at the expense of deferring the copy task to the system.

Programs written in an object-oriented language, such as Java, place new demands on checkpointing:

- Object-oriented programming style encourages the creation of many small objects. Each object may have some fields that are read-only, and others that are frequently modified. Thus, object encapsulation conflicts with programmer-based data placement strategies.
- The Java programmer has no control over the location of objects. Thus, it is impossible to ensure that frequently modified objects are all stored in the same page. Furthermore, a single page may contain both live objects and objects awaiting garbage collection.
- Java programs are run on a virtual machine which supports simultaneous processes. Since Java encourages parallelism as a software engineering method, libraries such as the GUI create many processes whose states are not always useful in a checkpoint. Also, object allocation is not usally managed on a per-process basis, thus adding unnecessary memory to a checkpoint. Finally, there is no simple solution to a transparent support of native methods.

These arguments suggest that a user-driven language-level approach may be appropriate for Java programs. Language-level checkpointing augments the source program with code to record the program state [16, 17, 26]. To promote safety, this checkpointing code should be introduced systematically, and interfere as little as possible with the standard behavior of the program. One approach is to add methods to each class to save and restore the local state. Checkpointing is then performed by a generic `checkpoint` method that invokes the checkpointing methods of each checkpointable object. Incremental checkpointing can be implemented by associating a flag with each object, indicating whether the object has been modified since the previous checkpoint. This checkpointing

*This research is supported in part by Bull, Alcatel, and by NSF under Grant EIA-9806718

†Author's current address: DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark

code can either be added manually or generated automatically using a preprocessor [16, 17]. In either case, localizing the code for saving and restoring the state of an object in its class definition respects encapsulation, thus enhancing overall program safety, and simplifies program maintenance.

Nevertheless, this generic programming model introduces overheads. First, because the `checkpoint` method is independent of the objects being checkpointed, it must interact with these objects using virtual calls. Virtual calls are less efficient than direct function calls, and block traditional compiler optimizations, such as inlining. Second, although the use of the modified flag reduces the size of checkpoints, it does not eliminate the need to visit each checkpointable object defined by the program.

This checkpointing strategy can be optimized by manually creating specialized checkpointing functions for recurring object structures in the program. When some of the objects are known not to be modified between specific checkpoints, all code relating to the checkpointing of those objects can be removed. Nevertheless, many specialized checkpointing routines may be needed, to account for the range of compound object structures used in different phases of the program. When the program is modified, these manually optimized routines may need to be completely rewritten. Thus, while these kinds of optimizations can yield significant performance improvements, performing them by hand is laborious and error-prone.

Our approach

In this paper, we propose to use *automatic program specialization* to automatically optimize a generic checkpointing algorithm based on programmer-supplied information about the fixed aspects of the object structure. Program specialization is a technique for automatically and aggressively optimizing a generic program with respect to information about the program inputs [11, 15]. This technique has been applied in a wide range of areas, including operating systems [20, 21, 31], and scientific programs [13, 23].

By specializing the checkpointing implementation with respect to recurring structural and modification patterns, we eliminate many tests, virtual calls, and traversals of unmodified data. Because specialization is automatic, these transformations can be performed reliably, and are simple to modify as the program evolves.

To assess the benefits of our approach in a realistic setting, we specialize the checkpointing of an implementation of a program analysis engine, which performs the kinds of analyses that are used in compilation or automatic program specialization. To analyze more precisely the benefits of our approach, we also consider a synthetic program in which we can vary the dimensions and modification pattern of the checkpointed structure. These results can be used as a metric to predict the benefits of specializing the checkpointing process for other applications. We have run the specialized programs on a 300 MHz Sun Ultra2 using the standard JIT of JDK 1.2.2, the HotSpot dynamic compiler, and the Harissa JVM which runs Java-to-C translated programs. We obtain the following results:

- Specializing with respect to the structure of a compound object optimizes the traversal of the subobjects by replacing virtual calls by inlined code.
- Specializing with respect to the modification pattern of a compound object eliminates tests and the traversal of completely unmodified objects.
- The program analysis engine example is divided into phases, each of which reads but does not modify the results of previous phases. We automatically generate a specialized checkpointing routine for each phase. Specializing with respect to both the object structure and the modification pattern gives speedups of up to 1.5 times.
- For the synthetic example, we first specialize with respect to the structure, and then with respect to both the structure and the modification pattern. Specialization with respect to the structure gives speedups up to 3. Specialization with respect to the structure and the modification pattern gives speedups proportional to the percentage of unmodified objects. When three quarters of the objects are unmodified, we obtain speedups up to 15.

The rest of this paper is organized as follows. We begin with a Java implementation of checkpointing, in Section 2. Section 3 then introduces program specialization and identifies opportunities for the specialization of the checkpointing implementation. Sections 4 and 5 assess the benefits of specialization of the checkpointing process on the program analysis engine and the synthetic example. Section 6 describes related work, particularly focusing on complementary approaches to language-level checkpointing. Finally, Section 7 concludes and suggests future work.

2 Incremental Checkpointing of Java Programs

We consider the checkpointing of an object-oriented program in which the state of the program can be recovered from the contents of the object fields. In this context, checkpointing amounts to recursively traversing the objects and recording the local state of each one; the stack is omitted. Similar strategies have been proposed by others, including Kasbekar *et al.* [16] and Killijian *et al.* [17].

2.1 Implementation

The implementation consists of the `Checkpointable` interface, which specifies the methods that must be provided by each object to be checkpointed, and a `Checkpoint` object, which drives the checkpointing process. These are defined in Figure 1. For simplicity, we assume that the checkpointed objects do not contain cycles. We also assume that checkpoints are constructed using a blocking protocol, and are written from the output stream to stable storage asynchronously.

Each checkpointable object contains a unique identifier and methods that describe how to record the state of the object and its children. Additionally, to implement incremental checkpointing, each object contains a flag indicating whether any fields of the object have been modified since the previous checkpoint. This functionality is captured by the `Checkpointable` interface. The unique identifier and the modification flag, which are defined in the same way for all checkpointable objects, are factored into a separate `CheckpointInfo` object, also defined in Figure 1.

```
public interface Checkpointable {
  public CheckpointInfo getCheckpointInfo();
  public void fold(Checkpoint c);
  public void record(OutputStream d);
}

public class Checkpoint {
  OutputStream d;

  public Checkpoint() {
    d = new OutputStream();
  }

  public void checkpoint(Checkpointable o) {
    CheckpointInfo info = o.getCheckpointInfo();
    if (info.modified()) {
      d.writeInt(info.getId());
      o.record(d);
      info.resetModified();
    }
    o.fold(this);
  }
}

public class CheckpointInfo {
  private int id;
  private boolean modified;
  public CheckpointInfo() {
    id = newId();
    modified = true;
  }

  // unique identifier
  public int getId() { return id; }
  private static int newId() { ... }

  // modification flag
  public boolean modified() { return modified; }
  public void setModified() { modified=true; }
  public void resetModified() { modified=false; }
}
```

Figure 1: Incremental checkpointing in Java

The `Checkpointable` interface specifies that each checkpointable object must define the methods `getCheckpointInfo()`, `record()`, and `fold()`. The method `getCheckpointInfo()` accesses the associated `CheckpointInfo` structure. The method `record(OutputStream d)` records the complete local state of the checkpointable object in the output stream `d`.[1] A value of base type is written directly, while a sub-object is represented by its unique identifier. The method `fold(Checkpoint c)` recursively applies the checkpointing object `c` to each of the checkpointable sub-objects.

Checkpointing is initiated by creating a `Checkpoint` object, which initializes the output stream. The user program then applies the `checkpoint` method to the root of each compound structure to record in the checkpoint. To implement incremental checkpointing, checkpointing of an object is divided into two steps. First, if the object has been modified, its unique identifier is recorded in the output stream, and its `record()` method is invoked to record its local state. The `modified` field is also reset. Then, regardless of whether the object has been modified since the previous checkpoint, the `fold` method of the object is invoked to recursively apply the checkpointing process to the children.

As in other approaches to checkpointing of object-oriented programs, the state of each object is restored from a checkpoint using a restore method local to the object. The definition of such a method is the inverse of the definition of `record`. The unique identifiers associated with each object are used to reconstruct the state from a sequence of incremental checkpoints. Because restoration is performed rarely, specialization seems unlikely to be interesting here.

2.2 Defining checkpointable objects

The methods required by the `Checkpointable` interface can be systematically defined either manually or automatically, as follows. A class implementing the `Checkpointable` interface creates a `CheckpointInfo` structure and defines the associated `getCheckpointInfo()` accessor function. Such a class also defines `record` and `fold` methods to record its local state and traverse its children, respectively. A class that extends a checkpointable class defines `record` and `fold` methods corresponding to its own local state. These methods invoke the respective methods of the parent class to checkpoint the inherited fields.

As an example, we use part of the implementation of the program analysis engine, presented in Section 4. Each phase of the program analysis engine stores its result in

[1] In practice, we instantiate `OutputStream` as a `DataOutputStream` composed with a `ByteArrayOutputStream`, as defined in the `java.io` package.

```
public abstract class Entry
  implements Checkpoint.Checkpointable {
  CheckpointInfo checkpointInfo =
    new Checkpoint.CheckpointInfo();

  public CheckpointInfo getCheckpointInfo() {
    return checkpointInfo;
  }
  public void record(OutputStream d) { }
  public void fold(Checkpoint c) { }
}

public class BTEntry extends Entry {
  BT bt;

  public void record(OutputStream d) {
    super.record(d);
    d.writeInt(bt.getCheckpointInfo().getId());
  }

  public void fold(Checkpoint c) {
    super.fold(c);
    c.checkpoint(bt);
  }

  // Other methods for manipulating the
  // BTEntry object
  ...
}
```

Figure 2: The `Entry` and `BTEntry` classes

a corresponding object. To capture the commonality between these objects, the class of each such object extends an abstract class `Entry`. The class `Entry` and an extension `BTEntry` are shown in Figure 2. The `Entry` class explicitly implements the `Checkpointable` interface. Thus, it creates the `CheckpointInfo` structure and defines the `getCheckpointInfo()` method. The `Entry` class also defines `record()` and `fold()` methods. These methods are trivial, because the `Entry` class has no local state. The `BTEntry` class inherits the `CheckpointInfo` structure of the `Entry` class. It defines its own `record()` and `fold()` methods, to carry out the checkpointing of its child `bt`. The `record()` method first invokes the `record()` method of the superclass, and then accesses the `CheckpointInfo` structure of the child to record the child's unique identifier. The `fold()` method first invokes the `fold()` method of the superclass, and then recursively applies the `checkpoint` method to the child.

3 Program Specialization

Program specialization is the optimization of a program based on supplementary information about its input. We first describe this technique, and then consider how to use it to optimize the checkpointing process.

3.1 Overview of program specialization

Program specialization optimizes a program to a specific usage context. This technique restricts the applicability of a program, in exchange for a more efficient implementation. Specialization of programs written in imperative languages, such as C and Fortran, achieves optimizations such as constant folding and loop unrolling [2, 3, 11]. Specialization of Java programs has been shown to reduce the overhead of data encapsulation, virtual calls, and run-time type and array-bounds checks [27]. Our implementation of checkpointing benefits from these optimizations.

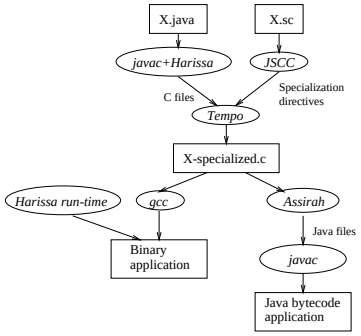

Figure 3: Structure of JSpec, specialization of class `X`

In the context of an object-oriented language, such as Java, the usage context can be described by *specialization classes* [32]. A specialization class describes how a class should be specialized, by declaring properties of the fields and methods of the specialized class. The declared methods are then specialized with respect to this information. Specialization classes are compiled by the *Java Specialization Class Compiler* (JSCC) into directives for the program specializer, and are thus not part of the program execution. We rely on the programmer to specify specialization classes that safely describe the execution context, just as we rely on the programmer to identify points at which specialized checkpointing can be useful.

We specialize the checkpointing process using the *JSpec* program specializer [27], illustrated in Figure 3. JSpec is based on *Tempo*, a program specializer for C [11]. To perform specialization of Java programs, the Java bytecode is first translated into C using the Harissa bytecode-to-C compiler [22]. The specialized C code can be compiled using any C compiler, and then executed in the Harissa JVM. At the C level, the specialized code can express optimizations of the virtual machine, such as the elimination of array-bounds checks, that cannot be expressed in Java. Alternatively, for portability, the specialized C code can be converted back to an ordinary Java program using the Harissa tool *Assirah*.

3.2 Specialization opportunities

The implementation of checkpointing offers two significant opportunities for specialization: specialization with respect to the structure of the checkpointed data and spe-

cialization with respect to the data modification pattern of the program. We now describe the benefits of these two kinds of specialization for the checkpointing process.

When there are recurring compound objects having the same structure, we can specialize the `checkpoint` method for this structure. Specialization replaces the virtual calls to the methods of the `Checkpointable` interface by direct calls. These direct calls can be inlined, or otherwise optimized by the compiler. Concretely, inlining generates a monolithic specialized checkpointing implementation for each compound object.

The use of the `modified` field can also be optimized by specialization. Suppose a program initializes a set of objects in one phase, and subsequently only reads their values. When this behavior can be determined before execution, the checkpointing process can be specialized to the fact that in the later phases the `modified` field of such objects is always `false`. This optimization eliminates the test in the `checkpoint` method, which in turn eliminates all reference to the `CheckpointInfo` structure. When combined with specialization to the structure of complex objects, this optimization can eliminate all traversal of compound objects that are completely unmodified between checkpoints.

4 A Realistic Application

Our approach to the optimization of checkpointing is targeted towards complex, long-running programs that manipulate many instances of similar compound structures. We achieve additional benefits when the program is organized in phases, each of which is known to modify only specific kinds of structures. We now describe such a program, a Java implementation of the analyses performed by the program specializer Tempo, and assess the benefits of specialization of the checkpointing process.

4.1 Overview of the program analysis engine

Effective program specialization demands precise, and often time-consuming, analyses. Following the structure of many compilers [1], these analyses are organized in phases, each of which uses, but does not modify, the results of the previous analyses. This kind of program can benefit from specialization of incremental checkpointing.

Concretely, we consider three of the analyses performed by Tempo: *side-effect analysis*, *binding-time analysis*, and *evaluation-time analysis*. Side-effect analysis determines the set of global variables read and written by each program statement. Binding-time analysis identifies expressions that can be evaluated using only the information available to the specializer [15]. Evaluation-time analysis ensures that variables referenced by the specialized program are properly initialized [14]. Our prototype implementation in Java of these analyses treats a simplified version of C.

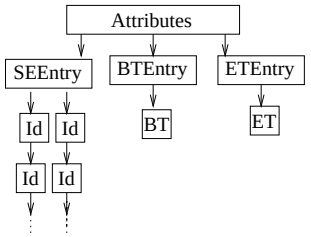

Figure 4: Organization of the `Attributes` structure

Each statement of the analyzed program is associated with an `Attributes` structure, which contains a field for the results of each phase of the analysis. Side-effect analysis collects sets of variables, while binding-time analysis and evaluation-time analysis each record only a single annotation. Thus, most of the information recorded in the `Attributes` structure comes from the side-effect analysis, and is fixed during subsequent phases. Consequently, specialization of the checkpointing process to eliminate the traversal of unmodified objects is most useful for the binding-time and evaluation-time analyses. These analyses are also typically longer than side-effect analysis, making checkpointing more desirable for these phases.

To treat loops, each analysis phase performs repeated iterations over the abstract syntax tree. At the end of each iteration, the local state is captured by the annotations stored at each node. Thus, the end of an iteration is a natural time at which to take a checkpoint.

4.2 Specialization opportunities

We now illustrate the specialization opportunities identified in Section 3.2 in the context of the program analysis engine. We specialize with respect to information about the `Attributes` structure, illustrated in Figure 4. Note that the `BTEntry` class was defined in Figure 2.

We first specialize the checkpointing implementation to the structure of an `Attributes` object. The specialization class `Checkpoint_Attributes` shown below declares that a specialized variant of the `checkpoint` method should be created for the `Attributes` class.

```
specclass Checkpoint_Attributes
  specializes Checkpoint {
  public void checkpoint(Checkpointable o),
    Attributes o;
}
```

Declaring such a specialization of the `Checkpoint` class for each class used in the program makes the types of the checkpointed objects explicit. Specialization replaces virtual calls by direct calls and field references. Virtual calls

```
checkpoint_attr(Checkpointable o) {
 Attributes attr = (Attributes)o;
 CheckpointInfo attrInfo = attr.getCheckpointInfo();
 if (attrInfo.modified()) {
  d.writeInt(attrInfo.getId());
  attr.record(d);
  attrInfo.resetModified();
 }
 SEEntry seEntry = attr.se;
 CheckpointInfo seEntryInfo = SEEntry.getCheckpointInfo();
 if (seEntryInfo.modified()) {
  d.writeInt(seEntryInfo.getId());
  seEntry.record(d); /* records both lists */
  seEntryInfo.resetModified();
 }
 BTEntry btEntry = attr.bt;
 CheckpointInfo btEntryInfo = btEntry.getCheckpointInfo();
 if (btEntryInfo.modified()) {
  d.writeInt(btEntryInfo.getId());
  btEntry.record(d);
  btEntryInfo.resetModified();
 }
 BT bt = btEntry.bt;
 CheckpointInfo btInfo = bt.getCheckpointInfo();
 if (btInfo.modified()) {
  d.writeInt(btInfo.getId());
  bt.record(d); /* virtual call */
  btInfo.resetModified();
 }
 ETEntry etEntry = attr.et;
 CheckpointInfo etEntryInfo = etEntry.getCheckpointInfo();
 if (etEntryInfo.modified()) {
  d.writeInt(etEntryInfo.getId());
  etEntry.record(d);
  etEntryInfo.resetModified();
 }
 ET et = etEntry.et;
 CheckpointInfo etInfo = et.getCheckpointInfo();
 if (etInfo.modified()) {
  d.writeInt(etInfo.getId());
  et.record(d); /* virtual call */
  etInfo.resetModified();
 }
}
```

Figure 5: Specialization of `checkpoint` w.r.t. the structure of an `Attributes` object

```
checkpoint_attr_btmodif(Checkpointable o) {
 Attributes attr = (Attributes)o;
 CheckpointInfo attrInfo = attr.getCheckpointInfo();
 if (info.modified()) {
  d.writeInt(attrInfo.getId());
  attr.record(d);
  attrInfo.resetModified();
 }
 BTEntry btEntry = attr.bt;
 CheckpointInfo btEntryInfo = btEntry.getCheckpointInfo();
 if (btEntryInfo.modified()) {
  d.writeInt(btEntryInfo.getId());
  btEntry.record(d);
  btEntryInfo.resetModified();
 }
 BT bt = btEntry.bt;
 CheckpointInfo btInfo = bt.getCheckpointInfo();
 if (btInfo.modified()) {
  d.writeInt(btInfo.getId());
  bt.record(d); /* virtual call */
  btInfo.resetModified();
 }
}
```

Figure 6: Specialization of `checkpoint` w.r.t. the modification properties of an `Attributes` object for the binding-time analysis

only remain for the methods of the `bt` (binding time) and `et` (evaluation time) objects, whose values are not known during specialization. Subsequent inlining and translation of the specialized C code back to Java produces the optimized implementation shown in Figure 5.

The program analysis engine also has the property that each phase only modifies its corresponding field of the `Attributes` structure. To describe this property, we define a specialization class indicating that the `modified()` method of the `CheckpointInfo` class should be specialized for the case where the `modified` flag has not been set to `true`. To create a specialized checkpointing implementation for use during the binding-time analysis phase, we define a specialization classes for the `Attributes`, `SEEntry`, and `ETEntry` classes indicating that the specialized `modified()` method should be used. The result of specializing according to these declarations is shown in Figure 6. The specialized checkpointing implementation for the evaluation-time analysis phase is specified similarly.

4.3 Performance assessment

Table 1 summarizes the performance of the checkpointing of the binding-time analysis and evaluation-time analysis phases. In a program specializer that treats full C, such as Tempo, these analyses can take up to several hours, depending on the complexity of the analyzed program. Since we treat a simplified version of C, the analyses are considerably faster. We have analyzed a 750-line image manipulation program: without checkpointing, binding-time analysis runs for 62.2 seconds and evaluation-time analysis runs for 6.4 seconds. We compare full checkpointing, incremental checkpointing, and specialized incremental checkpointing. A checkpoint is taken for each iteration of the analyses. The binding-time analysis requires nine iterations, while the evaluation-time analysis requires only three. For full checkpointing, we show the performance for the iterations with the minimum and maximum checkpoint sizes. For unspecialized and specialized incremental checkpointing, the checkpoints all have roughly the same size, so we give average figures. For the binding-time analysis phase, specialization gives speedups of over 1.3, and for the evaluation-time analysis phase specialization gives speedups of almost 1.5 over incremental checkpointing.

We have noted that specialization eliminates the traversal of unmodified objects. Thus, the traversal time represents the limit of the cost that can be eliminated by specialization. The last line of the table compares the traversal time for incremental and specialized incremental checkpointing. For the binding-time analysis phase, specialization reduces the traversal time by 1.8 times, and for the evaluation-time analysis phase specialization reduces the traversal time by over 2 times.

	Binding-time analysis (BTA)				Evaluation-time analysis (ETA)			
	full ckp. min. size	full ckp. max. size	incremental	specialized incremental (**speedup**)	full ckp. min. size	full ckp. max. size	incremental	specialized incremental (**speedup**)
Ckp. size (Mb)	12.52	21.88	1.40	1.40	11.04	11.17	0.55	0.55
Ckp. time (s)	5.3	9.08	1.34	1.00 (**1.34**)	4.56	4.65	0.71	0.48 (**1.49**)
Traversal time (s)	-	-	0.74	0.40 (**1.85**)	-	-	0.46	0.23 (**2.03**)

Table 1: Checkpoint size (in Mb) and execution time (in seconds). (JDK 1.2.2 JVM, Sun Ultra2 300MHz)

5 A synthetic application

To assess the benefits of our approach independent of a particular application, we consider a synthetic example, in which we can vary the structure of the checkpointed objects. The goal of these tests is to provide a metric for determining to what degree other applications can benefit from our approach. We consider checkpointing a set of compound structures, each containing five linked lists. We vary properties of these structures such as the length of the lists, the percentage of modified list elements, and the number of integer-typed fields stored in each list element.

The test program constructs 20,000 compound structures, randomly chooses constituent list elements to be modified according to the constraints of the experiment, and performs a single checkpoint. Our benchmarks present the time to construct the checkpoint. Unless otherwise stated, the Java programs were translated to C before specialization and then run in the Harissa JVM.

We first compare incremental checkpointing to full checkpointing. When some objects are not modified, incremental checkpointing reduces the cost of recording the current state. Nevertheless, incremental checkpointing also introduces tests into the traversal of the compound structures. Figure 7 shows that even when all of the objects are modified the added cost is negligible. The speedup obtained by incremental checkpointing increases as the number of modified objects decreases, and as the cost of recording the state of each object increases. When only a quarter of the objects are modified, and when 10 integers are recorded for each modified object, incremental checkpointing is over 3 times faster than full checkpointing.

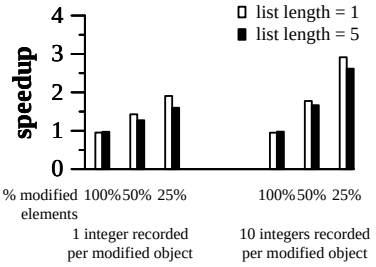

Figure 7: Incremental checkpointing (Harissa JVM)

Incremental checkpointing reduces the number of objects recorded in the checkpoint, but still requires a complete traversal of the compound structures to identify modified objects. Specialization with respect to properties of the object structure optimizes the traversal. In particular, we specialize with respect to the following structural information.

- The shape of the compound structures.
- The set of lists that may contain modified objects.
- The positions in these lists where a modified object may occur.

The speedups with respect to incremental checkpointing achieved by these specialization opportunities are summarized in Figures 8 through 11. The percentages in each figure indicate the percentage of possibly modified objects that are actually modified. For example, in Figure 9, in which the number of lists that contain modified objects is constrained, the case labelled “50%” where there are three modified lists means that among those three lists, half of the objects have been modified since the previous checkpoint.

Specialization with respect to the shape of each compound structure eliminates virtual calls and permits inlining. These optimizations give the most speedup when there are few modified objects, and thus the cost of the structure traversal dominates. Figure 8 shows that the speedup as compared to unspecialized incremental checkpointing ranges from 1.5 when all objects are modified and 10 integers are written for each modified object, to over 3 when each list has length 5, only 25% of the objects are modified, and only one integer is written for each modified object.

When some lists are known to be completely unmodified, specialization with respect to this information eliminates the traversal of such lists. Here, the greatest speedup is obtained when there are long lists, of which few may contain modified objects (see Figure 9). For lists of length 5, when only one value is recorded for each modified object, the speedup ranges from 2 to 9, as the number of lists that may contain modified objects decreases. When 10 integers are recorded for each modified object, the speedup is reduced by up to half.

Specializing with respect to the specific positions within each list at which modified objects can occur eliminates the need to test the other objects. We consider the case where a modified object can only occur as the last element of each

list. This is the worst case, because only tests, but not object traversals, are eliminated. Because the number of eliminated tests depends on the length of the lists, we achieve the best speedup for long lists. Figure 10 shows that for lists of length 5, when only one value is recorded for each modified object, the speedup over unspecialized incremental checkpointing ranges from 5 to 15, depending on the number of lists that may contain modified objects. When 10 integers are recorded for each object, these speedups range from 2 to 11.

So far, we have assessed the performance of specialized C code. For portability, we can also translate the specialized C code back to Java using the Assirah tool. In our third specialization experiment above (c.f. Figure 10), we specialize with respect to both the number of lists that may contain a modified object and the position at which a modified object may occur in each list. Figure 11 compares the performance of the Java specialized code with the performance of the unspecialized Java implementation of incremental checkpointing, for lists of length 5. As shown in Figure 11a, using the JDK 1.2.2 JIT compiler, we obtain speedups of up to 12. As shown in Figure 11b, combining JDK 1.2.2 with the state-of-the art dynamic compiler HotSpot, we obtain speedups of up to 6 over the performance of the unspecialized code, also running on HotSpot. As shown in Table 2, the Harissa code is significantly faster than the code produced by the JDK 1.2.2 JIT compiler or HotSpot. Table 2 also shows that the unspecialized code run with HotSpot can be faster than the specialized code run without HotSpot. Thus, one may wonder whether HotSpot subsumes program specialization. Nevertheless, Figure 11b shows that the specialization further improves performance under HotSpot, demonstrating that specialization and dynamic compilation are complementary.

6 Related work

Automatic program-transformation techniques have already been used to improve the reliability and performance of source-level checkpointing. The C-to-C compilers c2ftc and porch, developed by Ramkumar and Strumpen [26, 30] and by Strumpen [29], respectively, add code around each procedure call to enable a program to manage the checkpointing and recovery of its control stack. A preprocessor in the Dome system provides a similar facility for parallel C++ programs [5, 6]. Plank *et al.* propose to use data-flow analysis to determine automatically, based on hints from the user, the regions of memory that are not modified between checkpoints [4, 24]. Calls to functions in a checkpointing library (libckpt for Sparc or CLIP for Intel Paragon) are then automatically inserted into the source program. Killijian *et al.* and Kasbekar *et al.* use compile-time reflection provided by OpenC++ [10] to add check-

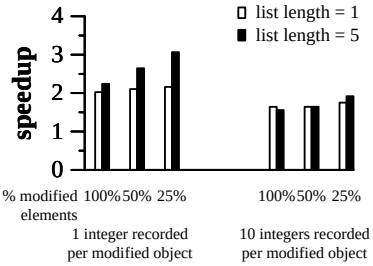

Figure 8: Specialization w.r.t. the object structure (Harissa JVM)

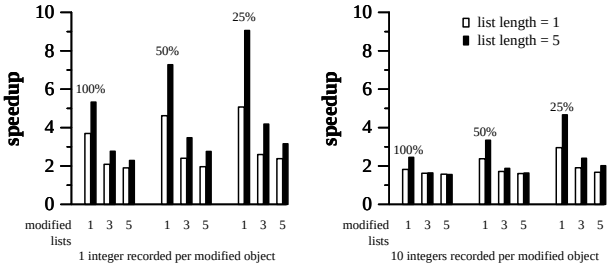

Figure 9: Specialization w.r.t. the object structure and the number of lists that may contain modified elements (Harissa JVM)

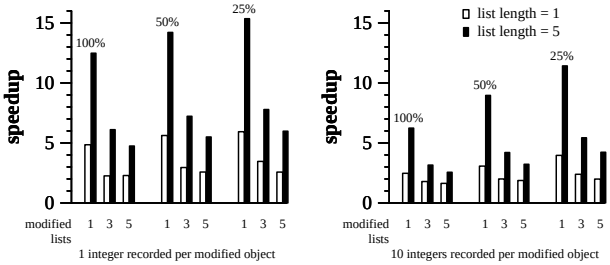

Figure 10: Specialization w.r.t. the object structure and the number of lists whose last element can be modified (Harissa JVM)

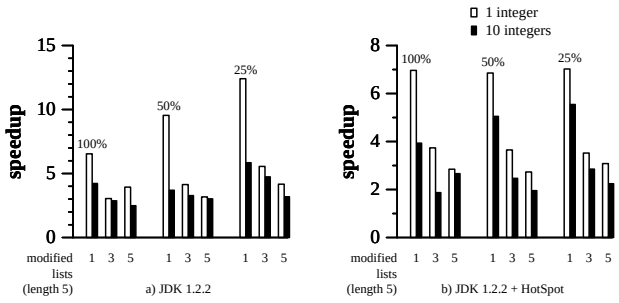

Figure 11: Specialization w.r.t. the object structure and the number of lists whose last element can be modified (Sun JVM)

	Possibly mod. lists	Harissa			JDK 1.2.2			JDK 1.2.2 + HotSpot		
		100%	50%	25%	100%	50%	25%	100%	50%	25%
Unspecialized code	1	1.05	0.98	0.95	3.99	1.98	1.76	1.80	1.56	1.32
	5	1.80	1.36	1.14	10.92	7.05	4.03	4.51	2.41	1.71
Specialized code	1	0.17	0.10	0.08	0.95	0.54	0.30	0.46	0.31	0.24
	5	0.70	0.42	0.27	4.39	2.33	1.27	1.70	1.23	0.76

Table 2: Checkpoint execution time (in seconds), 10 integers written for each element

pointing code at the source level to the definitions of C++ objects [16, 17]. The reflection-based approaches are most closely related to ours. Essentially, we use program specialization to optimize checkpointing methods of the form generated by reflection.

Several of these source-level approaches address the problem of incremental checkpointing. The analysis proposed by Plank *et al.* to detect unmodified regions of memory is performed at compile time, and is thus necessarily approximate. The reflective approach of Killijian *et al.* associates a modification flag with each object field. Maintaining and testing these flags at run time adds substantial overhead: extra space to store the modification flags, extra time on every assignment to update the associated flag, and extra time during checkpointing to test the flags. Our approach exploits both compile-time and run-time information. When it is possible to determine at compile time that an object is not modified between checkpoints, specialization eliminates the code to save the state of the object. When it is not possible to determine this information at compile time, the modified flag is retained in the specialized program and tested at run time. Because specialization is automatic, it is feasible to create many implementations, to account for the modification patterns of each phase of the program, without changing the source code.

Language-level checkpointing for Java provides independence from the virtual machine. Other approaches have simplified the checkpointing process and reduced checkpoint size by omitting aspects of the underlying language implementation. The Stardust [9] and Dome [5, 6] systems for SIMD parallelism in heterogeneous environments restrict checkpointing to synchronization points in the `main` function, eliminating the need to record the stack. In the context of Java, Killijian *et al.* also record only object fields, and thus omit the stack [17].

Checkpointing is conceptually similar to *serialization*, the conversion of an object structure into a flat representation. In Java, serialization is implemented using run-time reflection. Reflection is used both to determine the static structure of each object (its type, field names, etc.), and to access the recorded field values. The structure of an object, however, does not change during execution. Thus, repetitively determining this information at run time is inefficient. Braux and Noyé propose to eliminate the overheads of Java reflection using program specialization [8]. These techniques could useful in extending our approach to a checkpointing implementation based on reflection.

Several Java-based mobile agent systems use serialization to transmit the state of an agent to another host [18, 33]. The Concordia system also provides extensive checkpointing facilities based on serialization to recover from transmission failures [33]. Specialization of the checkpointing process could improve the performance of these systems as well.

7 Conclusion and future work

We have shown that automatic program specialization can significantly improve the incremental checkpointing of Java programs. Because specialization is automatic, the generated code is correct. This approach has several advantages: (i) multiple checkpoint procedures can be generated for a single program, permitting to exploit per-phase modification patterns, and (ii) checkpointing can be implemented straightforwardly to facilitate program evolution and maintenance, without sacrificing performance.

This work can fit into a series of automated tools to improve the performance of language-level checkpointing. In the approach we have presented, the user must identify which compound structures are used frequently in the program, and the regions in which such structures are not modified. To automate this process, we propose to automatically construct specialization classes based on an analysis of the data modification pattern of the program. If we additionally use reflection as proposed by Kasbekar *et al.* and by Killijian *et al.* to automatically generate the checkpointing methods for each class [16, 17] and automatically modify the source code as proposed in the `c2ftc` and `porch` systems to save and restore the stack [26, 29, 30], we obtain an efficient and transparent language-level implementation of checkpointing for Java programs.

Acknowledgments

We thank Ulrik Pagh Schultz and Miroslav Malek for helpful comments, and James Plank for shepherding the final version of this paper. We also thank the other members of the Compose group who participated in the design and the implementation of JSpec.

Availability

Examples described in this paper are available at `http://www.irisa.fr/compose/jspec/checkpoint`. Tempo, Harissa, JSpec and the Java Specialization Class Compiler are available at the Compose web page `http://www.irisa.fr/compose/`.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[3] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[4] M. Beck, J.S. Plank, and G. Kingsley. Compiler-assisted checkpointing. Technical Report CS-94-269, University of Tennessee, December 1994.

[5] A. Beguelin, E. Seligman, and E. Stephan. Application level fault tolerance in heterogeneous networks of workstations. Technical Report CMU-CS-96-157, School of Computer Science, Carnegie Mellon University, August 1996.

[6] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[7] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[8] M. Braux and J. Noyé. Towards partial evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, MA, USA, January 2000. ACM Press.

[9] G. Cabillic and I. Puaut. Stardust: an environment for parallel programming on networks of heterogeneous workstations. *Journal of Parallel and Distributed Computing*, 40:65–80, February 1997.

[10] S. Chiba. A metaobject protocol for C++. In *OOPSLA'95 Conference Proceedings*, pages 285–299, Austin, TX, USA, October 1995. ACM Press.

[11] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[12] E.N. Elnozahy, D.B. Johnson, and W. Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[13] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[14] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2), 2000.

[15] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[16] M. Kasbekar, S. Yajnik, R. Klemm, Y. Huang, and C.R. Das. Issues in the design of a reflective library for checkpointing C++ objects. In *18th IEEE Symposium on Reliable Distributed Systems*, pages 224–233, Lausanne, Switzerland, October 1999.

[17] M.-O. Killijian, J.-C. Fabre, and J.-C. Ruiz-Garcia. Using compile-time reflection for object checkpointing. Technical Report Noo99049, LAAS, February 1999.

[18] D.B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998.

[19] G. Muller, M. Banâtre, N. Peyrouze, and B. Rochat. Lessons from FTM: an experiment in the design & implementation of a low cost fault tolerant system. *IEEE Transactions on Reliability*, pages 332–340, June 1996. Extended version available as IRISA report 913.

[20] G. Muller, R. Marlet, and E.N. Volanschi. Accurate program analyses for successful specialization of legacy system software. *Theoretical Computer Science*, 248(1–2), 2000.

[21] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[22] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[23] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[24] J.S. Plank, M. Beck, and G. Kingsley. Compiler-Assisted Memory Exclusion for Fast Checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*, Winter 1995.

[25] J.S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software—Practice and Experience*, 29(2):125–142, 1999.

[26] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 58–67, Seattle, WA, June 1997. IEEE.

[27] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.

[28] L.M. Silva and J.G. Silva. An experimental evaluation of coordinated checkpointing in a parallel machine. In J. Hlavicka, E. Maehle, and A. Pataricza, editors, *Proceedings of The Third European Dependable Computing Conference (EDCC-3)*, volume 1667 of *Lecture Notes in Computer Science*, pages 124–139, Prague, Czech Republic, September 1999. Springer.

[29] V. Strumpen. Compiler technology for portable checkpoints. http://theory.lcs.mit.edu/ strumpen/porch.ps.gz, 1998.

[30] V. Strumpen and B. Ramkumar. *Fault-Tolerant Parallel and Distributed Systems*, chapter Portable Checkpointing for Heterogeneous Architectures, pages 73–92. Kluwer Academic Press, 1998.

[31] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.

[32] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.

[33] T. Walsh, N. Paciorek, and D. Wong. Security and reliability in concordia(tm). In *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS'98)*, pages 44–53, Kona, Hawaii, January 1998. IEEE Computer Society Press.