# Efficient Incremental Checkpointing of Java Programs

Julia L. Lawall, Gilles Muller

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Efficient Incremental Checkpointing of Java Programs*

Julia L. Lawall and Gilles Muller

**N°3810**

Novembre 1999

THÈME 2


ISSN 0249-6399 ISRN INRIA/RR--3810--FR+ENG

# Efficient Incremental Checkpointing of Java Programs

Julia L. Lawall and Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n° 3810 — Novembre 1999 — 23 pages

**Abstract:** This paper presents a user-driven language-level approach to the checkpointing of Java programs. First, we describe how to systematically associate incremental checkpoints with Java classes. While being safe, the genericity of this solution induces a substantial execution overhead. Second, to solve the dilemma between genericity and performance, we use automatic program specialization to transform the generic checkpointing procedures into highly optimized ones. Specialization exploits two kinds of information: (i) structural properties about the program classes, (ii) knowledge of unmodified data structures in a specific program phase. The latter information allows us to generate phase-specific checkpointing procedures. 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 object structure and the modification pattern. Measured speedups are up to 15.

**Key-words:** incremental checkpointing, Java, program specialization

*(Résumé : tsvp)*

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

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

# Création de points de reprises incrémentaux efficaces pour les programmes Java

**Résumé :** Cet article présente une approche utilisateur à la création de points de reprise pour les programmes Java. Tout d'abord nous décrivons comment associer de manière systématique des points de reprise incrémentaux à une classe Java. Bien que sûre, cette solution entraîne un surcoût d'exécution substantiel en raison de sa généricité. Afin de résoudre le conflit entre généricité et performance, nous proposons d'utiliser la spécialisation automatique de programmes pour transformer les méthodes génériques de création de points de reprise en des méthodes spécifiques optimisées. La spécialisation tire profit de deux sortes d'informations : (i) des propriétés structurelles sur les classes du programme, (ii) la connaissance de la non-modification de structures de données lors d'une phase d'exécution précise du programme. Nous avons évalué notre approche sur deux programmes de test, une application réaliste consistant en un moteur d'analyse de programmes, et une application synthétique servant de métrique. Nos résultats montrent que la spécialisation apporte un gain en temps d'exécution proportionnel à la complexité de la structure objet du programme et au canevas de modification des objets. Sur nos expérimentations, nous avons mesuré un gain d'un facteur allant jusqu'à 15 sur le temps d'exécution de la sauvegarde d'un point de reprise.

**Mots-clé :** points de reprise incrémentaux, Java, spécialisation de programmes

# 1 Introduction

Checkpointing is known to introduce overhead proportional to the size of the checkpoint [12, 26]. 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*, in which system-level facilities are used to keep track of the modified virtual-memory pages [7, 18]. Each checkpoint contains only the pages that have been modified since the previous checkpoint.

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 can contain both live objects, and objects awaiting garbage collection.
- Java programs are run on a virtual machine. Some internal state of the virtual machine is not needed to reconstruct the state of the program. A system-level approach cannot easily make this distinction.

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, 24]. In this respect, the most important issue is 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 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 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 could 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. This optimization eliminates the need to traverse objects that are completely unmodified between checkpoints. 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 information about the fixed aspects of the object structure. Program specialization is a technique for automatically and aggressively optimizing a program with respect to user-supplied information about the program inputs [11, 15]. Automatic program specialization has been applied in a wide range of areas, including operating systems [19, 20], networks [29], and scientific programs [13, 22]. Recently, automatic program specialization has been implemented for Java [25]. While program specialization is a general tool and has a wider application scope than checkpointing, the need for safety and the genericity in object structure make specialization a very appropriate technique for optimizing checkpointing. Additionally, using automatic program specialization, it is feasible to safely generate a specialized checkpointing implementation for each phase of the program. To our knowledge, the study presented in this paper is the first attempt to use program specialization in fault tolerance.

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. Specialization of Java programs is driven by user-defined *specialization classes* [30], auxiliary declarations that correspond to the class structure of the program, but specify particular values for object fields and identify methods to specialize accordingly. Specialization classes also document the optimizations performed by the automatic tool, 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 specialization the checkpointing process for other applications. We obtain the following results:

- Specializing with respect to the structure of a compound object optimizes the traversal of the sub-objects 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 in Section 2 by defining an implementation of checkpointing in Java. Section 3 then introduces program specialization and identifies opportunities for the specialization of the checkpointing implementation. In Section 4, we then present a realistic example program of the kind that can benefit from our techniques, and assess the speedup of the checkpointing process obtained by specialization. Next, in Section 5, we investigate the speedups obtained for a synthetic example, which permits to assess the benefits obtainable using our approach on a wider range of programs. Section 6 describes related work, particularly focusing on complementary approaches to language-level checkpointing. Finally, Section 7 concludes and presents perspectives for 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 fields of the objects. In this context, checkpointing amounts to recursively traversing the objects and recording the local state of each one. Similar strategies for checkpointing object-oriented programs have been proposed by others, including Kasbekar *et al.* [16] and Killijian *et al.* [17].

## 2.1 The 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 written from the output stream to stable storage asynchronously.

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

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: An implementation of checkpointing in Java

Associated with each checkpointable object are 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 the object has 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, which is also specified in Figure 1.

The `Checkpointable` interface specifies that each checkpointable object should define the methods `record()`, `fold()`, and `getCheckpointInfo()`. The method `record(OutputStream d)` records the state of the checkpointable object in the output stream `d`. A value of base type can be written directly, while a sub-object can be referred to using 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 `checkpoint` method is then applied to the root of each object structure to record in the checkpoint. To implement incremental checkpointing, checkpointing is carried out in 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 Using checkpointable objects in an object-oriented application

The programmer can systematically define the methods required by the `Checkpointable` interface, using the following strategy. A class explicitly 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. The program analysis proceeds in phases, each of which stores its result in 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.

## 2.3 Assessment

The strategy for checkpointing that we have presented is systematic, and thus enhances safety. Nevertheless, the implementation is hard to optimize even with state-of-the-art compilers such as Sun's HotSpot[1].

- `Checkpointable` is declared as an interface, so that arbitrary objects can be declared as checkpointable, independent of the inheritance hierarchy. Nevertheless, the use of an interface implies that a checkpointable object's `checkpointInfo` field must be accessed using a method call, rather than a direct field reference.
- When a program defines many checkpointable classes, the class of the checkpointed object cannot be determined when compiling the `checkpoint` method. Thus, the uses of the `record()`, `fold()`, and `getCheckpointInfo()` methods must be implemented as virtual calls, which are difficult to optimize.

[1] Available from `URL: http://java.sun.com/products/hotspot/`

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

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

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

- Within a single invocation of the `checkpoint` method, there are several virtual calls to methods of the same object. The number of virtual calls could be reduced by shifting more of the checkpointing code into the user classes, but at the cost of code duplication and reduced maintainability.
- Incremental checkpointing is implemented by testing the `modified` field of each object. While this approach is precise, it introduces many branches, which are slower than straight-line code.

These inefficiencies can be eliminated by creating specific checkpointing routines for each kind of object in the program. In the next section, we show that we can safely generate such optimized implementations using automatic program specialization.

# 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 [25]. Our implementation of checkpointing benefits from these optimizations.

In the context of an object-oriented language, such as Java, the usage context can be described by *specialization classes* [30]. A specialization class describes how a class should be specialized, by declaring properties of a subset of the fields and methods of the specialized class. A field or method parameter can be declared to have a more restricted type than the original declaration, or can be declared to have a specific value. 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 specialize the checkpointing process using the program specializer *Tempo* for C programs, developed by the Compose group at IRISA [11]. Tempo has been adapted to perform specialization of Java programs, by first translating Java bytecode into C using the Harissa bytecode-to-C compiler [21], and then specializing the resulting C code [25]. The specialized C code can be compiled using any C compiler, and then executed in the Harissa run-time environment. 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*. This approach to the specialization of Java programs is illustrated in Figure 3.

## 3.2 Specialization opportunities in incremental checkpointing

The implementation of checkpointing offers two significant opportunities for specialization: specialization with respect to the structure of the checkpointed data and specialization 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 to 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 replaces the call to `getCheckpointInfo()` by a direct field reference, and eliminates the overhead of calling separate `record()` and `fold()` methods.

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

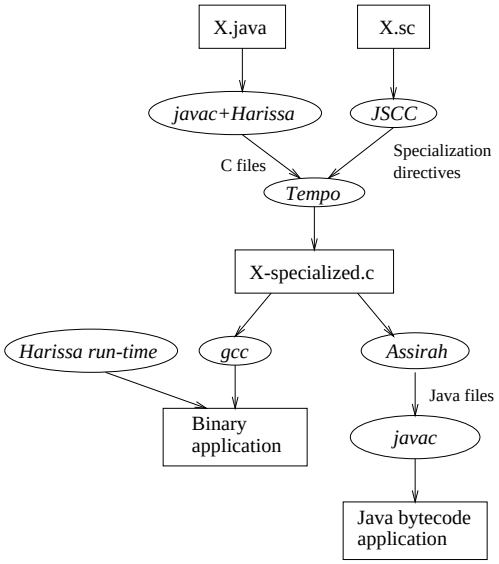


Figure 3: Structure of the prototype, specialization of class `X`

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, which no longer needs to be accessed. When combined with specialization to the structure of complex objects, this optimization produces straight-line code. Together, these optimizations can eliminate all traversal of complex 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 opportunities for specialization of the checkpointing process.

## 4.1 Overview of the program analysis engine

Effective program specialization demands precise, and often time-consuming, analyses. Following an organization common to many compilers [1], these analyses are organized in

phases, each of which uses, but does not modify, the results of the previous analyses. Furthermore, each phase simply adds information to a fixed attribute structure. 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 that are referred to in the specialized program are properly initialized [14]. Our prototype Java implementation of these analyses treats a simplified version of C.

Each statement of the 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 recursive programs, each analysis phase performs repeated iterations over the abstract syntax tree. 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. At the end of each iteration over the abstract syntax tree, 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 in checkpointing the program analysis engine

We now illustrate the specialization opportunities identified in Section 3.2 in the context of the checkpointing of the implementation 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 only remain for the methods of the `bt` (binding time) and `et` (evaluation time) objects, whose values are not known during specialization.

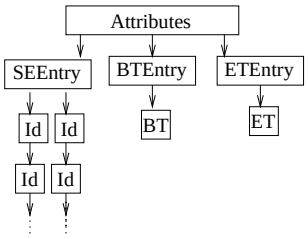


Figure 4: Organization of the `Attributes` structure

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. Figure 6 shows some of the specialization classes used to indicate that during the binding-time analysis phase only the `bt` field of the `Attributes` structure can be modified. `CheckpointInfo_unmodified` declares that a specialized instance of the `CheckpointInfo` class should be created for objects that are never modified. This specialization class is used in the specialization classes `SEEntry_unmodified` and `BTEntry_unmodified` to declare that the corresponding objects are not modified. These two classes are then used by `Attributes_only_bt_modified` to declare that its `se` and `et` fields are unmodified. Finally, `Checkpoint_BT` indicates that the `checkpoint` method should be specialized to such objects. The result of specializing according to these declarations is shown in Figure 7. The specialized method is a refinement of the result of specializing with respect to the structure, shown in Figure 5. Only the code pertaining to the checkpointing of `bt` remains.

Specialization for the other phases proceeds similarly. For the evaluation-time analysis, we also specialize with respect to the fact that for some nodes of the abstract syntax tree, the entire attribute structure is never modified.

## 4.3 Performance assessment

We now assess the performance of the specialized checkpointing code. Tests were performed on a 300 MHz Sun Ultra2. We translate the specialized C code back to Java and measure its performance using the HotSpot dynamic compiler. The final paper will also contain benchmarks for Harissa, and for the standard JIT of JDK 1.2.2.

Table 1 summarizes the performance of the checkpointing of the binding-time analysis and evaluation-time analysis phases. The analyzed program is a 750-line image manipulation program. 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

```
checkpoint_attr(Checkpointable o) {
  Attributes attr;
  SEEntry seEntry;
  BTEntry btEntry; BT bt;
  ETEntry etEntry; ET et;
  CheckpointInfo attrInfo, seEntryInfo, btEntryInfo, btInfo, etEntryInfo, etinfo;

  attr = (Attributes)o;
  attrInfo = attr.getCheckpointInfo();
  if (attrInfo.modified())
    {
      d.writeInt(attrInfo.getId());
      attr.record(d);
      attrInfo.resetModified();
    }
  seEntry = attr.se;
  seEntryInfo = SEEntry.getCheckpointInfo();
  if (SEEntryInfo.modified())
    {
      d.writeInt(seEntryInfo.getId());
      seEntry.record(d); /* records both lists */
      seEntryInfo.resetModified();
    }
  btEntry = attr.bt;
  btEntryInfo = btEntry.getCheckpointInfo();
  if (btEntryInfo.modified())
    {
      d.writeInt(btEntryInfo.getId());
      btEntry.record(d);
      btEntryInfo.resetModified();
    }
  bt = btEntry.bt;
  btInfo = bt.getCheckpointInfo();
  if (btInfo.modified())
    {
      d.writeInt(btInfo.getId());
      bt.record(d); /* virtual call */
      btInfo.resetModified();
    }
  etEntry = attr.et;
  etEntryInfo = etEntry.getCheckpointInfo();
  if (etEntryInfo.modified())
    {
      d.writeInt(etEntryInfo.getId());
      etEntry.record(d);
      etEntryInfo.resetModified();
    }
  et = etEntry.et;
  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

```
specclass CheckpointInfo_unmodified specializes CheckpointInfo {
  modified = false;
  public boolean modified();
}

specclass SEEntry_unmodified specializes SEEntry  {
  CheckpointInfo_unmodified checkpointInfo;
  IdList_unmodified read, written;
  public void fold(Checkpoint c);
  public CheckpointInfo getCheckpointInfo();
}

specclass ETEntry_unmodified specializes ETEntry {
  CheckpointInfo_unmodified checkpointInfo;
  ET_unmodified et;
  public void fold(Checkpoint c);
  public CheckpointInfo getCheckpointInfo();
}

specclass Attributes_only_bt_modified specializes Attributes {
  SEEntry_unmodified se;
  ETEntry_unmodified et;
  public void fold(Checkpoint c);
  public CheckpointInfo getCheckpointInfo();
}

specclass Checkpoint_BT specializes Checkpoint {
  public void checkpoint(Checkpointable o), Attributes_only_bt_modified o;
}
```

Figure 6: Specialization classes for the binding-time analysis phase

```
checkpoint_attr_btmodif(Checkpointable o) {
  Attributes attr; BTEntry btEntry; BT bt;
  CheckpointInfo attrInfo, btEntryInfo, btInfo;

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

Figure 7: Specialization of `checkpoint` w.r.t. the modification properties of an `Attributes` object

| | BTA | | | | ETA | | | |
|---|---|---|---|---|---|---|---|---|
| | full Ckp. min. size | full Ckp. max. size | inc. | specd. inc. (**speedup inc.**) | full Ckp. min. size | full Ckp. max. size | inc. | specd. inc. (**speedup inc.**) |
| Ckp. size (Kb) | 12524 | 21881 | 1408 | 1408 | 11039 | 11177 | 551 | 551 |
| Ckp. time (ms) | 5300 | 9082 | 1346 | 1006 (**1.34**) | 4565 | 4652 | 713 | 480 (**1.49**) |
| Traversal time (ms) | - | - | 739 | 400 (**1.85**) | - | - | 462 | 228 (**2.03**) |

Table 1: Checkpoint size (in Kb) and execution time (in ms). (Sun JVM)

minimum and maximum checkpoint sizes. For unspecialized and specialized incremental checkpointing, all of the checkpoints 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. As noted in Section 4.2, evaluation-time analysis provides additional specialization opportunities, because the `Attributes` fields of some kinds of expressions are never modified.

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.

The speedups obtained here are smaller than those presented in the next section for the synthetic application. There, specialization is applied to the entire checkpointed structure. For the program analysis engine, however, other objects are included in the checkpoint. The traversal of these objects is not optimized by specialization with respect to the `Attributes` structure.

# 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 the degree to which other applications can be expected to benefit from our approach. We consider a compound object containing five sub-objects, each of which is a linked list. We vary properties of this structure such as the length of the lists, the percentage of modified objects, and the number of integers that are recorded for each modified object.

The test program constructs 20,000 compound objects, and performs a single checkpoint. Our experiments were carried out on a 300MHz dual-processor Sun Ultra2. We use the checkpointing implementation of Figure 1. `OutputStream` is instantiated as a `DataOutputStream` composed with a `ByteArrayOutputStream`, as defined in the `java.io` package. The Java programs were translated to C using Harissa before specialization. In our first set of experiments, the specialized C code was compiled using `gcc` version 2.8.1 at optimization level `O3`, and then executed in the Harissa environment. In our last experiment, we translate the specialized code back to Java, and measure its performance using JDK 1.2.2, with the standard Just-In-Time (JIT) compiler and with the HotSpot dynamic compiler. In each case, we consider only the time to construct the checkpoint. Our experiments show that specialization can improve the performance of incremental checkpointing by up to 15 times, depending on the complexity of the object structure and the percentage of unmodified objects.

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 objects. Figure 8 shows that despite the additional tests, when all of the objects are modified, incremental checkpointing and full checkpointing have essentially the same performance. Incremental checkpointing gives the most speedup when there are few modified objects, and when the cost of recording the state of each modified object is relatively high. Incremental checkpointing is over 3 times faster than full checkpointing when only a quarter of the objects are modified, and when 10 integers are recorded for each modified object.

Incremental checkpointing reduces the number of objects recorded in the checkpoint, but still requires a complete traversal of the object structure 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 structure of the compound objects.
- The set of lists that may contain modified objects.
- The positions in these lists where a modified object can occur.

The speedups with respect to incremental checkpointing achieved by these specialization opportunities are summarized in Figures 9 through 12. The percentages in each figure indicate the percentage of objects that are modified.

Specialization with respect to the structure of each compound object eliminates virtual calls and permits inlining. These optimizations give the most speedup when there are few modified objects, and thus the cost of the object traversal dominates. As shown in Figure 9, 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 a quarter 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, and when little data is recorded for each modified object. The speedup obtained over incremental checkpointing is summarized in Figure 10. 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 most speedup for long lists. Figure 11 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 11), we specialize with respect to both the number of lists that may contain a modified object and the position at which a modified object can occur in each list. Figure 12 compares the performance of the Java specialized code with performance of the unspecialized Java implementation of incremental checkpointing, for lists of length 5. As shown in Figure 12a, using the JDK 1.2.2 JIT compiler, we obtain speedups of up to 12. As shown in Figure 12b, combining JDK 1.2.2 with the state-of-the art dynamic

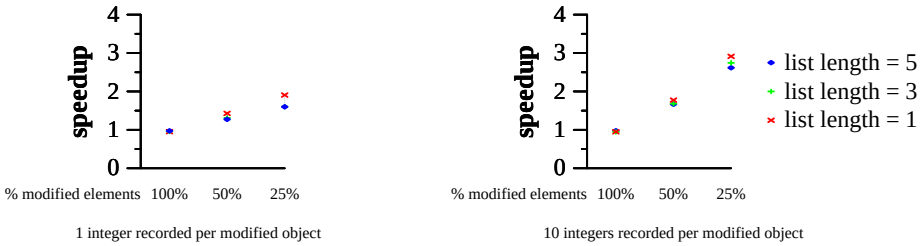


Figure 8: Incremental checkpointing (Harissa JVM)

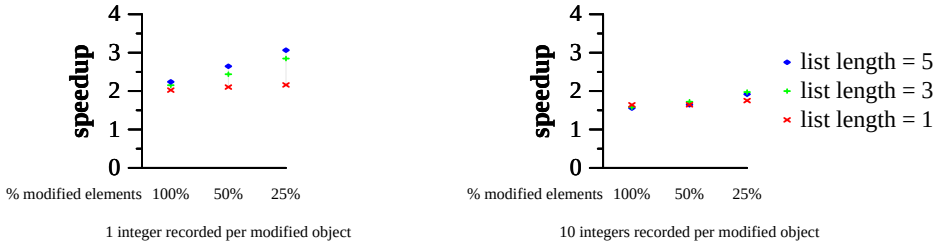


Figure 9: Specialization with respect to the object structure (Harissa JVM)

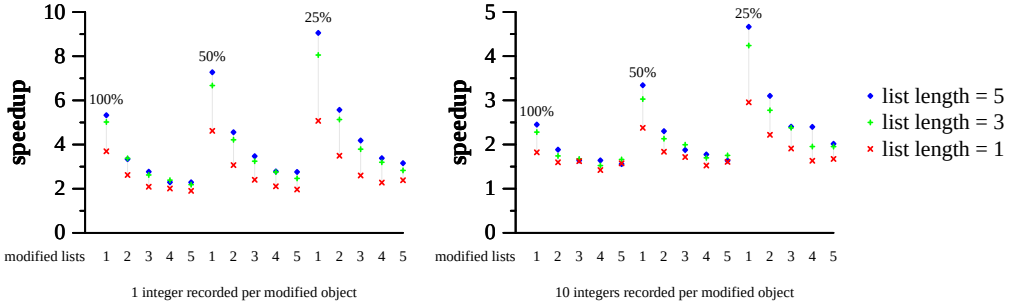


Figure 10: Specialization with respect to the object structure and the number of lists that may contain modified elements (Harissa JVM)

| | 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 | 1048 | 977 | 948 | 3993 | 1980 | 1759 | 1809 | 1560 | 1325 |
| | 5 | 1804 | 1364 | 1137 | 10918 | 7053 | 4036 | 4516 | 2412 | 1711 |
| Specialized code | 1 | 168 | 109 | 83 | 946 | 536 | 301 | 460 | 309 | 239 |
| | 5 | 705 | 423 | 269 | 4391 | 2334 | 1267 | 1697 | 1235 | 763 |

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

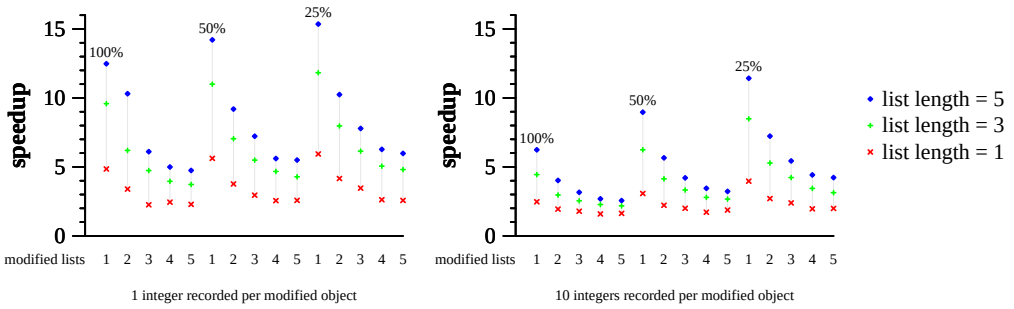


Figure 11: Specialization with respect to the object structure and the number of lists whose last element is modified (Harissa JVM)

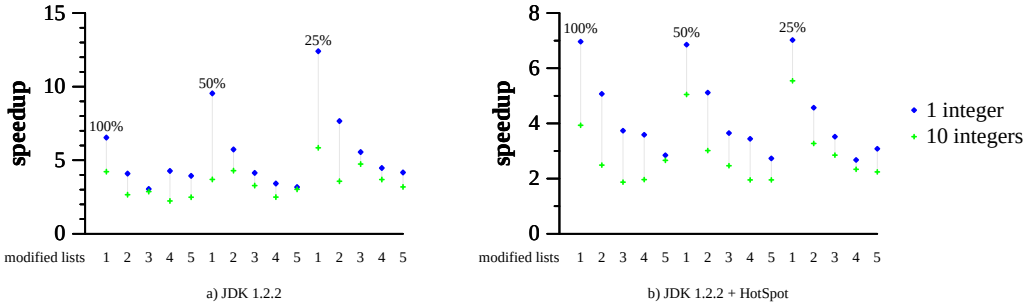


Figure 12: Specialization with respect to the object structure and the number of lists whose last element is modified (Sun JVM)

compiler HotSpot, we obtain speedups of up to 6 over the performance of the unspecialized code, also running on HotSpot. As shown by 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 12b 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 [24, 28] and by Strumpen respectively [27], add code around each procedure call that enables 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, 23]. Calls to functions in a checkpointing library (libckpt for Sparc or CLIP for Intel Paragon) are then automatically inserted into the source program according to the results of the analysis. Killijian *et al.* and Kasbekar *et al.* use compile-time reflection provided by OpenC++ [10] to add checkpointing code at the source level to the definitions of C++ objects [16, 17]. These approaches are most closely related to ours. Essentially, we use program specialization to optimize checkpointing methods of the form they generate 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 carried out 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 in order 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 to be 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.

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.[2] In the context of Java, Killijian *et al.* also record only object fields, and thus omit the stack [17]. By comparing the checkpointing of a recursive Java program using their approach with the checkpointing of a comparable C program using porch, they found that porch has a much higher checkpoint overhead, because of recording the stack.

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 has proposed 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.

# 7 Conclusion and future work

In this paper, we have shown that automatic program specialization can significantly improve the incremental checkpointing of Java programs. Because specialization is carried out automatically, 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 sacrifying performance.

This work opens up many possibilities for further research. 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 develop an analysis that identifies phases of the program in which particular structures are not modified. Specialization classes could be automatically constructed based on this information. 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 [24, 27, 28], we obtain an efficient and transparent language-level implementation of checkpointing for Java programs.

**Acknowledgments**

We wish to thank Ulrik Pagh Schultz for his numerous comments and careful proof-reading. We also thank the other members of the Compose group who participate to the design and the implementation of our Java specializer.

[2] As noted above, Dome also provides a preprocessor that allows checkpoints including the stack to be taken elsewhere in the program.

### Availability

Selected examples from this paper are available at `http://www.irisa.fr/compose/jspec/checkpoint`. Tempo, Harissa, and the Java Specialization Class Compiler are available through the Compose web page `http://www.irisa.fr/compose/`. A first prototype of specialization of Java programs using Tempo, including Assirah, will be available in December 1999.

## 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. Speeding up the Java serialisation framework through program transformation. In *E-COOP'99 Workshop - Object Technology for Product-line Architectures*. European Software Institute, June 1999.

[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. to appear.

[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] 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 research report 913.

[19] 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. To appear.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399