System Checkpointing using Reflection and Program Analysis

John Whaley

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
`jwhaley@alum.mit.edu`

Abstract. This paper describes a technique for checkpointing a running system by a combination of reflective introspection and program analysis. By using an extension to Java's Reflection API which allows activation frames and other aspects of execution state to be reflectively inspected and modified, we can halt at and restart from arbitrary points in the execution. We apply this checkpointing technique to an area that is not typically associated with reflection — optimization of memory footprint and startup time. We have successfully used this technique in the `joeq` virtual machine to reduce the heap size and the application startup time significantly.

1 Introduction

The Java programming language has gained a significant following due to a number of reasons, including its clean, object-oriented design and automatic storage management. It is a very popular choice for server-side applications; however, on the client side, Java has yet to make significant headway against more established languages like C and C++.

There are currently two major problems that must be overcome before Java can become prevalent on the client side. The first is application startup time [5]. Applications using the Swing GUI interface can take on the order of minutes to start up. This is due to a number of factors, including large amounts of inefficient initialization code in the class libraries and the fact that (in most virtual machines) the code is recompiled from class files on every execution.

The second major stumbling block in the adoption of Java on the client side is memory footprint. Even simple GUI applications can regularly consume 20 megabytes or more of memory [8]. What is more, in most systems this memory is not shared among applications, which means that 20 megabytes *per running instance* are consumed. Again, this is due to a number of factors, including the liberalism of virtual machine garbage collectors in allocating memory and the dynamic and object-oriented nature of the Java language. Because of dynamic features such as reflection and dynamic class loading, the system cannot perform whole-program analysis to eliminate the (as-of-yet) unused fields, methods, and other runtime data structures.

To attack these problems, we developed a general technique of checkpointing the state of the system using reflection. Checkpointing is the act of saving the state of a running system such that it can be recreated later in time. Checkpointing has many applications, including system recovery, debugging, bootstrapping, and process migration. We use checkpointing for a different purpose. By running the system up to a certain point and then checkpointing it, we can use the checkpointed version to avoid startup time costs; in essence, application startup time becomes bounded only by disk latency. Furthermore, we can perform analysis during the checkpointing process to optimize the system, reducing the memory footprint by eliminating unnecessary code, objects, fields, classes, and runtime data structures.

Using reflection makes our technique very flexible. Because we use reflection, we can precisely control the checkpointing process, deciding what data to ignore and what data needs to be reinitialized on virtual machine startup. Furthermore, the technique is entirely virtual machine and architecture independent; we actually use this technique to bootstrap the `joeq` virtual machine across platforms [14].

The remainder of the paper is organized as follows. Section 2 outlines the checkpointing technique and associated algorithms. Section 3 presents data on the effectiveness of the technique in reducing code and heap size and improving startup time in our implementation in the `joeq` virtual machine [14]. Section 4 discusses related work, and we conclude in Section 5.

2 Technique

Our basic checkpointing technique is as follows:

1. Start the virtual machine and let it execute until a desired point. Suspend all threads.
2. Determine the root set. If necessary, inspect the activation records (stack traces) of the threads.
3. Given the root set, determine the necessary parts of the system (code and data) and any code necessary to reconstruct the necessary state.
4. Serialize the necessary code and data to a file, in the standard format for the virtual machine.
5. On subsequent executions, simply map the file to memory and execute the reconstruction code to continue execution from the given point.

Our current implementation uses a very simple flow-insensitive, context-insensitive type-based pointer analysis to determine the necessary code and data. Namely, any field that is accessed, any method that is called, and any class that is instantiated is considered to reach all program points. Section 2.1 describes the algorithm in more detail.

After the necessary code and data are determined, we serialize the code and data in the standard format for the virtual machine. We also include relocations for every code and data reference, so that the code and data segments can be relocated between executions. Section 2.3 gives details on this process.

2.1 Algorithm description

```
doAlgorithm() {
  method_worklist = buildRootSet();
  while (!method_worklist.isEmpty()) {
    Method m; InstructionIndex a;
    (m, a) = method_worklist.pull();
    analyzeReachableCode(m, a);
  }
}
```

Fig. 1. The checkpointing algorithm

Pseudocode for the algorithm can be found in Figure 1. The algorithm first initializes a worklist of <*Method, InstructionIndex*> pairs using the root set, and then analyzes each of the methods in the worklist. There are two techniques to obtain the root set. The first technique, which is portable, is to have it specified by the programmer, *e.g.* the main method or event loop of a program. The second technique, which requires some virtual machine extensions, is to build the root set using the stack traces of the running threads.

```
addObject(Object o) {
  if (table.contains(o)) return;
  HeapAddress a = allocateSpace(getObjectSize(o));
  table.add(o, new Pair(o, a));
  addType(o.getClass());
  forall (Field f = o.getClass().fields()) {
    if (f.isReferenceType() && isNecessaryField(f)) addObject(f.get(o));
  }
}
```

Fig. 2. Pseudocode for adding an object

Figure 2 contains the pseudocode for marking an object as necessary. To keep track of the necessary objects, we use a hash table keyed on the identity hash code of the objects. This hash table maps from the object to a pair: <*object, address*>. *object* is a reference back to the object; *address* refers to the address of the object in the output image.

When an object that has never been encountered before is added, we reserve space for it in the output image, generate an <*object, address*> pair for it, and register the pair in the hash table. We also mark the object's type as instantiated. Then, for each reference field of the object that has been marked as necessary, we add the object referenced by that field.

```
analyzeReachableCode(Method m, InstructionIndex a) {
  Worklist w = new Worklist(); w.add(a);
  while (!w.isEmpty()) {
    Instruction i = w.pull();
    forall (Field f = i.getAccessedFields()) addField(f);
    forall (Class c = i.getInstantiatedTypes()) addType(c);
    forall (Method m2 = i.getTargetMethods()) addMethodToWorklist(m2, 0);
    forall (Instruction j = i.successors()) w.add(j);
  }
}
```

Fig. 3. Pseudocode for analyzing the reachable code from a given starting point

Figure 3 contains pseudocode for analyzing the reachable code from a given starting point. We traverse the instructions using a worklist algorithm. For each instruction, any field that can be accessed is added to the list of necessary fields. Likewise, if an instruction can instantiate an object, we add the type to the set of instantiated types. For call instructions, we add all target methods to the method worklist. Finally, we add the indices of all possible successor instructions in the current method to the worklist.

```
addType(Class c) {
  if (classes.contains(c)) return; classes.add(c);
  addObject(c);
  forall (Method m = visited_methods) {
    if (c.overrides(m)) addMethodToWorklist(c.getMethod(m), 0);
  }
}
```

Fig. 4. Pseudocode for marking a type as instantiated

Figure 4 gives the pseudocode for marking a type as instantiated. When a new type is encountered, we add the `Class` object, which ensures that the vtable and other run time structures will be correctly allocated in the image. We then add the implementations of any necessary overridden method to the worklist.

Figure 5 lists the pseudocode for adding a field as necessary. When adding a new reference (non-primitive) instance field to the necessary set, we need to add any objects that are reachable via that field on already visited objects, so we iterate through the set of necessary objects that contain the newly added field and use reflection to get their values. In the case of reference static fields, we always add their values.

```
addField(Field f) {
  if (fields.contains(f)) return; fields.add(f);
  if (f.isReferenceType()) {
    if (f.isInstanceField()) {
      forall (Object o = necessaryObjects()) {
        if (o.getClass().containsField(f)) addObject(f.get(o));
      }
    } else addObject(f.get(null));
  }
}
```

Fig. 5. Pseudocode for marking a field as necessary

2.2 Generating startup code

Some objects contain values that are specific to the particular execution instance, or require some code to be executed on startup. For example, memory needs to be reallocated and file handles need to be reopened. Therefore, the algorithm also calculates the code to be executed during virtual machine startup to reinitialize the execution-specific data values.

Execution-specific data values are determined in one of two ways. The first way is by using the type of the object. Certain fields in objects always refer to data that is execution instance specific and therefore must be reinitialized on virtual machine startup. When reflectively inspecting these fields, we keep track of their instances so that we can generate the correct reinitialization code. In some cases, it is not possible to reconstruct the state using only the object fields. In such cases, we run the application using instrumented versions of the initialization routines which cache the incoming parameters. To perform reinitialization, we simply call those routines again with the same parameters. Some data values that are execution-specific cannot be determined solely by their type. For example, system properties are stored as `String`s in a `Vector` object, so we cannot determine simply from their type that they should be reinitialized at run time. We explicitly enumerate such values along with the code necessary to reinitialize them.

In some cases, special application knowledge is necessary for correct checkpointing. If code has executed that was dependent on some value that may change between executions, we may run into problems. In such cases, the programmer must manually specify the application data to be reinitialized along with the code necessary to perform the reinitialization. However, in many applications such code is not necessary [9].

2.3 Serializing the code and data

After we have determined the set of necessary objects, methods, fields, and classes, we serialize them to a file using the standard object format for the virtual machine. We iterate through all of the entries in the hash table by starting

	Fields Before	Fields After	Objects Before	Objects After	Heap Size Before	Heap Size After	Code Size Before	Code Size After
joeq	3840	2590	474231	326539	16690524	11029044	939981	774637
javac	5674	3882	587061	432073	20752496	14829948	1310719	1138206
SwingSet	13496	11491	1040864	687935	38333768	24642388	2157767	1745599
Forte	19348	15698	3832894	1812932	64398534	38304308	3639503	3048391

Fig. 6. Savings in heap memory

	Original Startup time	Checkpoint Startup time
joeq	27.2 s	1.7 s
javac	35.7 s	1.9 s
SwingSet	140.7 s	3.6 s
Forte	241.6 s	5.9 s

Fig. 7. Savings in startup time

address, using reflection to examine each of their fields and writing the object at the specified address. For each non-null object reference that we encounter, we also store relocation information so that it can be successfully relocated. Likewise, we generate code for all of the necessary methods, adding relocations where necessary.

3 Results

This section presents some experimental results from our implementation in the `joeq` virtual machine [14]. In the results, `joeq` refers to the "application" of the virtual machine itself. `javac` refers to Sun's `javac` compiler. `SwingSet` is the example Swing application contained in the standard Java distribution. `Forte` is the Forte for Java integrated development environment. Applications were executed to the start of their main methods and checkpointed at that point.

Table 6 shows the reduction in heap and code size after our technique. As we can see from this data, our technique is very effective in reducing object counts and heap size, up to a 39% reduction. It also significantly reduces code size, up to 19%.

Table 7 shows the reduction in startup time due to our technique. The times given in the table are wall-clock times until the entry point of the application. We rebooted between runs to avoid caching effects. The first column is the time it takes to start up the application normally. The second column gives the time it takes to start up the application from the checkpointed file. All timings were taken when executing on the `joeq` virtual machine. We were able to attain huge savings on application startup time with our technique. The checkpoint version was, in essence, bounded by disk time. Later executions (once the checkpoint file was in the disk cache) only took a fraction of a second.

4 Related work

The technique described in this paper of system checkpointing using execution-state reflection has similarities to many prior techniques. In this section, we compare and contrast our technique to related work.

Our overall checkpointing technique is similar to a proposal called Orthogonal Persistence for the Java platform (OPJ) [2]. Orthogonal persistence is an approach to making application objects persist between program executions. To assist in recreating state, similar to our technique OPJ uses *restart callbacks*, which allow the programmer to specify code that will be executed on restart. The major difference between our technique and OPJ is that because OPJ is only concerned with persisting individual objects, it requires more support from the application program. It does not perform any kind of program analysis; it simply uses reachability. Furthermore, OPJ requires the use of a special virtual machine, whereas our technique can use the built-in Java Reflection API.

Our technique is similar to the technique used by the Jalapeño virtual machine to bootstrap itself [1]. However, the Jalapeño technique does not include any support for the serialization of execution state, nor does it support reinitializing state at run time. Also, it does not include any sort of program analysis to determine the extent of the checkpointing.

There has been other work on checkpointing in the context of migrating applications [10], using extra processors for fault tolerance [12], post-mortem and replay debugging, elimination of boundary condition errors [13], etc. Our work is very similar to user-level transparent checkpointing techniques [11]. Such techniques usually work by compiling the application program with a special checkpointing library. Our technique, on the other hand, relies on program analysis and therefore can optimize the result for size and speed. Our system also shares many of the same restrictions as user-level checkpointers; for example, programs should not cache certain types of data [9].

Other researchers have attacked Java's large memory usage and long startup times in other ways. Systems like Echidna [4], rheise os [6], and others [3] allow multiple "processes" to run inside a single virtual machine. This allows the virtual machine startup time and the memory consumption for the base virtual machine to be amortized across each process.

The analysis technique of using the program state to optimize code has similarities to the well-known technique of partial evaluation [7]. The analysis as described here is very simple and only makes minimal use of the available information. Much more extensive use of the data to perform true partial evaluation optimizations is possible.

5 Conclusion

Checkpointing has many useful applications, such as system recovery, debugging, bootstrapping, and process migration. This paper introduces a new use for checkpointing — improving startup time and memory footprint. Our checkpointing technique uses a combination of reflection and program analysis to determine the necessary parts of the program, and serializes only those parts. By using reflection, our technique is very general and powerful. We can easily and precisely adjust what data we checkpoint and automatically generate reinitialization code for data that must be reinitialized on every execution.

We implemented this technique in the `joeq` virtual machine and presented performance results that show that even with a very simple flow-insensitive and context-insensitive program analysis, this technique can be very effective in reducing application startup time and memory footprint.

There are many further opportunities for taking advantage of the extra information about the program that is available at checkpoint time. For example, by using partial evaluation, we can optimize the application based on values that are not known at compile time, but known at the checkpoint time. A higher level example is to examine the elements of a hash table and use them to derive a perfect hash function. Such techniques will improve not only startup time and heap size, but also overall execution time.

References

1. B. Alpern. Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.
2. M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, Dec. 1996.
3. G. Czajkowski. Application isolation in the Java virtual machine. In *Proceedings of OOPSLA-00*, pp. 354–366, Oct. 15–19 2000.
4. L. Gorrie. Echidna http://www.javagroup.org/echidna, 1998.
5. E. Gun, S. Arthur, J. Gregory, and B. Bershad. A practical approach for improving startup latency in Java applications, In Proceedings of the Workshop on Compiler Support for Systems Software, Atlanta, Georgia, May 1999.
6. R. Heise. rheise.os. http://www.progsoc.uts.edu.au/r̃heise/projects/rheise.os
7. N. Jones, C. Gomard, and P. Sestoft. Partial evaluation and automatic program generation. Prentice Hall. 1993.
8. Tornado Labs. Java 3D benchmark results. http://www.tornadolabs.com/News/BenchJ3d_Results/benchj3d_results.html, 2000.
9. M. Litzkow and M. Livny. Making workstations a friendly environment for batch jobs. In Proc. 3rd Wks. on Work. Oper. Sys., April 1992.
10. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the Condor distributed processing system, 1997.
11. J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Tech Report UT-CS-97-372, 1997.
12. J. S. Plank, Y. Kim, and J. J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pp. 351–360, 1995.
13. Y. M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *Proc. 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23)*, pp. 138–144, Toulouse, France, 1993.
14. J. Whaley. joeq virtual machine. http://joeq.sourceforge.net, 2001.