Private Memory Allocation Analysis for Safety-Critical Java

Andreas E. Dalsgaard
Department of Computer Science
Aalborg University
andrease@cs.aau.dk

René Rydhof Hansen
Department of Computer Science
Aalborg University
rrh@cs.aau.dk

Martin Schoeberl
Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Safety-critical Java (SCJ) avoids garbage collection and uses a scope based memory model. This memory model is based on a restricted version of RTSJ [3] style scopes. The scopes form a clear hierarchy with different lifetimes. Therefore, references between objects in different scopes are only allowed from objects allocated in scopes with a shorter lifetime to objects allocated in scopes with a longer lifetime.

To ensure memory safety, programmers are required to either manually annotate the application with complex annotations, rely on a runtime test of each reference assignment, or statically analyze all reference assignments and avoid runtime checks when all assignments are proven to be correct. A violation of the assignment rule at runtime leads to an unchecked exception. For safety-critical code that needs to be certified, runtime exceptions must be avoided and the absence of illegal reference assignments needs to be proven. In this paper we present a static program analysis tool that automates the proof that no illegal assignments occur.

1. INTRODUCTION

The standard defining safety-critical Java (SCJ) [6] introduces the concept of *private memories*. Private memories are based on RTSJ style scoped memories, but are only accessible by a single thread of control—hence the name private memory. These private memories are used for temporary objects. An initial private memory is entered on each release of a handler. All objects are reclaimed at the end of the release. For further flexibility, private memories can be nested. To share data among handlers, there exists immortal memory and mission memory. Together with the private memories, these memory areas form a hierarchy with different life times.

One important consequence of the scoped memory model of SCJ is that without garbage collection, the lifetime of an object is now the responsibility of the programmer as it is no longer handled automatically (and safely). Instead it is determined directly by the lifetime of the scope it lives in, which again is controlled by the programmer. While enabling more flexibility and affording the programmer better control of memory use, it also opens the door for dangling references: a reference to an object that no longer exists.

In order to avoid *dangling references*, references within this hierarchy are only allowed from a shorter lived memory (an inner memory) towards an object in a longer lived memory (an outer scope). In SCJ the problem of dangling references is handled either by manually annotating applications with detailed memory scope information or dynamically with runtime checks. Using annotations requires an analysis for verifying the correctness of the annotations. Handling the problem dynamically can be done by checking, at runtime, that all operations on objects are based on a valid, non-dangling reference and throwing an exception if not. Since the SCJ memory hierarchy does not allow the RTSJ style cactus stack of scopes [3], each scope can be assigned a level, which simplifies the assignment check [8]. However, even when the assignment check is simplified, having runtime exceptions is considered very problematic for any system that is not supposed to terminate and especially so for safety-critical real-time systems that need certification.

In this paper we present an approach using program analysis to prove the absence of illegal assignments for objects living in SCJ style memory areas. The analysis is a context-sensitive points-to analysis for Java bytecode where a stack of SCJ memory scopes is used as contexts, enabling a straightforward check for potential violations of memory safety. A WALA [1] based prototype of the analysis has been implemented. Our implementation is targeting level 0 and 1 SCJ applications. With this implementation it is possible to present problems in the code, where illegal assignments might occur at runtime, to the programmer.

The paper is organized as follows: the following section gives background on the safety-critical Java specification. Section three and four present the memory safety analysis and implementation. Section five give results of experiments with the memory analysis. Section six, seven and eight present related work, future work and how to get access to the source code, respectively. Section nine concludes.

2. SAFETY-CRITICAL JAVA

Safety-critical Java (SCJ) [6] is intended for future safety-critical systems that need certification. To allow certification of Java programs only a very restricted subset of Java is defined. SCJ itself is based on the real-time specification for Java (RTSJ) [3]. It is a subset of RTSJ with some ad-

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

ditional class files. It is so defined that it can in principle be implemented on top of RTSJ. That is also the way the reference implementation (RI) is provided. However, it has to be noted that the combination of the SCJ on top of RTSJ is not a preferred combination for certification. If the implementation of the RTSJ is part of the safety-critical system, it needs to be certified as well. Therefore, one can just use the full RTSJ.

The SCJ specification is developed within the Java community process (JCP) under specification request number JSR 302. To cover different criticality levels, SCJ defines three different levels with increasing complexity of implementations and increasing expressive power for the application programmer. Level 0 provides a single-threaded cyclic executive. All memory areas (immortal, mission, and private) are available in level 0. As individual executions of handler releases are not preempted, the backing store for the private memory of a handler can be reused by the next handler. Level 1 introduces preemptive scheduling with ceiling based locks. Furthermore, interrupt handlers, written in Java, are allowed in level 1. Level 2 provides the notion of nested missions for more dynamic systems. It is possible to keep part of the system running and starting and stopping other parts during runtime. Level 2 also introduces an adapted version of RTSJ's NoHeapRealtimeThread.

Concurrence is represented as *handlers* in SCJ, similar to RTSJ style event handlers. In fact the SCJ handlers are a subclass of RTSJs BoundAsyncEventHandler. These handlers are either periodic or event triggered.

2.1 Missions and Scheduling

SCJ has the notion of missions. An application can consist of several missions, where each mission might represent a different operation mode. The mission itself consists of the handlers and the mission memory. The handlers within a mission are created int he initialization phase and the number of handlers is fixed for a mission. Handlers come in two flavors: a periodic event handler to be released time-triggered and an aperiodic handler released by an event. The event to release an aperiodic handler can be a software event or an interrupt.

A mission consists of three phases: initialization, execution, and cleanup. In the initialization phase the mission memory is created by the SCJ implementation and all handlers and data created during initialization is by default allocated in the mission memory. Data shared between handlers must be allocated in mission memory. Data shared between missions must be allocated in immortal memory.

The SCJ application is started on the transition from the initialization to the execution phase. During the execution phase no new handlers can be registered or started. Temporal objects are allocated in the handler's private memory. Allocation in mission memory is not prohibited, but strongly discouraged. After the cleanup phase, the mission memory is cleared and a new mission can be started.

An SCJ application is represented by a class that implements Safelet and at least one class that extends Mission. Simple programs, consisting of a single mission, can use one class the extends Mission and implements Safelet.

2.2 The Memory Model

Three different memory areas are available for an SCJ application: immortal memory, mission memory, and private memories. Immortal memory is the same as immortal memory in the RTSJ. It contains static fields, objects that are created during class initialization, and application data that needs to be preserved over mission boundaries. Mission memory, as the name implies, exists as long as a mission is active (in any of the three phases).

Within this paper the most interesting memory area is private memory. Each handler has an initial private memory, which is cleared after a release has finished. Therefore, no data can survive individual releases. To allow more flexibility within the release of a handler, the handler can enter nested private memories. In practise, the feature is only useful when a nested private memory is entered more than once per release. Otherwise the temporary data could also be allocated in the initial private memory. The object representing a private memory needs to be reused when repeatedly entering private memory to avoid leaking memory. However, nested private memories might be sized differently. Therefore, the implementation of the private memory area needs to be able to resize a memory area [9]. The private scopes are by intention private and are entered by a static method.

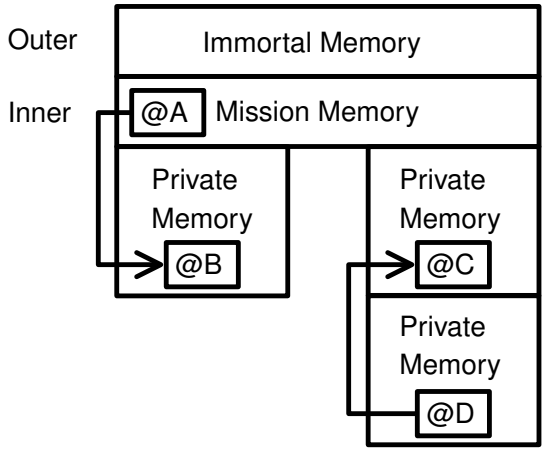

Figure 1: Memory scopes of an SCJ application.

In Figure 1 the memory scopes of an example SCJ application are shown. The example application implements a single mission with two event handlers. Each event handler has a private memory area and thereby a separate scope. The event handler on the right has entered another layer of private memory. The small boxes denoted `@A`, `@B`, `@C`, and `@D` represent objects allocated in a particular scope, represented by the immediately surrounding box. An arrow between two objects represents a reference to the object at the arrow head, stored in a field in the object at the arrow base. The reference from object `@A` to object `@B` is prohibited by the scope assignment rules of the SCJ as Mission Memory will have a dangling reference when the Private Memory is cleaned.

During execution of an SCJ application the scopes in use by the application, at any point in time, can be viewed as a *stack* of scopes, see Figure 1 for an example. When talking about a stack of scopes we will call the first scope on the stack (from the bottom of the stack) the outer scope of the next scopes which we will refer to as the inner scopes. As

an example, the right-most event handler in Figure 1 has a scope stack consisting of: Immortal Memory, Mission Memory, Private Memory, and finally Nested Private Memory at the top. For SCJ applications Mission Memory will always be the outer scope of the Private Memory scope of an event handler and potential nested Private Memory scopes.

3. MEMORY SAFETY ANALYSIS

In this section we discuss a sound memory safety analysis for SCJ that can statically guarantee an SCJ application will never (attempt to) violate the memory scope rules. In addition to the increased assurance and lower maintenance cost achieved by eliminating potential bugs early in development [2], it also makes it possible to dispense with the runtime checks otherwise necessary to enforce memory safety [8, 15]. Such runtime checks are undesirable in safety-critical real-time applications, not only due to the added runtime overhead, but also because of the inherent difficulties in handling memory safety violations at runtime without negatively impacting the scheduling of critical tasks. While this may also be true for annotation-based methods, the extra time and resources spent on developing and maintaining annotations (and potential code duplication [14]) may negate some or all of the positive effects.

In the following we describe a fully automated static memory safety analysis, that can verify that a program does not violate memory safety or point the programmer to where such violations may potentially occur.

In order to find potential violations of the memory scope rules, the memory safety analysis must find all references that may be stored in a field on an object in a given scope, to an object allocated in an inner scope. For this it needs to determine, for all objects, which scope they are allocated in as well as track when the current scope changes (see below). This makes points-to analysis a good basis for the memory safety analysis (as also noted in [11]), since a sound points-to analysis computes a conservative (over-)approximation of all the possible object references that may be stored in any field or local variable, which is exactly what is needed for the memory safety analysis. Extending the basic points-to analysis to a context-sensitive analysis using memory scopes as contexts, essentially tagging each abstract object reference in the analysis with the scope in which it was allocated, makes it almost trivial to find potential violations or, indeed, guarantee that there are no violations.

For the purpose of the analysis we will define a scope as:

$$\mathsf{Scope} = \mathsf{MemoryAreaID} \times \mathsf{ScopeType}$$

where MemoryAreaID is a unqiue identifier for the memory area represented by the scope and ScopeType is the type of scope defined by:

$$\mathsf{ScopeType} = \{\mathsf{IMMORTAL}, \mathsf{MISSION}, \mathsf{PM}, \mathsf{UNKNOWN}\}$$

where the first three correspond to immortal, mission, and private memory respectively. However, for the analysis we must also be able to represent the situation where, for some reason, it is not possible to determine the current scope (corresponding to the 'top' value in the underlying abstract analysis domain). We will refer to this as an *unknown memory*.

Based on these definitions we can define a scope stack as:

$$\mathsf{ScopeStack} = \mathsf{Scope}^*$$

As mentioned above, In our context-sensitive pointer analysis, we use scope stacks as contexts. This means both object references and objects are tagged with a scope stack. To be able to identify if a object reference is in an outer scope we define an ordering on the set of scope stacks:

$$ss_1 \sqsubseteq ss_2 \quad \text{iff} \quad \exists ss_2' : ss_2 = ss_2' :: ss_1$$

in other words, a scop stack ss_1 is less than scope stack ss_2 if and only if the stack ss_1 is a postfix of the stack ss_2.

Furthermore we note that equallity between scopes only is true if the MemoryAreaID and ScopeType are identical except if any of the scope types is UNKNOWN in which case the scopes are not equal.

As the SCJ API contains a number of functions that may be used to directly change the current memory area, the memory safety analysis must track the use of such functions in addition to the (extended) points-to analysis:

handleAsyncEvent
: is the function called by the scheduler when an event handler is activated. When handleAsyncEvent is called a private memory area of the event handler is entered and the scope should therefore be changed.

enterPrivateMemory
: is used to enter a private memory scope. It takes an object of a class implementing the Runnable interface as an argument. When the run() method of the runnable object is called it is executed in a new private memory scope.

getCurrentManagedMemory
: is used to get a reference to the current scope in the form of a ManagedMemory object. This reference can be passed around and is typically used to pass the reference of the current scope to an inner scope in a variable. From the inner scope the reference stored in a variable can be used to pass a result out of the inner scope.

executeInArea
: is called on a MemoryArea or ManagedMemory object with a runnable object as argument, it is possible to pass a result out of an inner scope. The runnable object needs to be initialised and in the run() method the result can be copied to an outer scope. E.g., the getCurrentManagedMemory method can be used to get a ManagedMemory object.

getMemoryArea
: can be used to get a reference to a MemoryArea. The method is similar to getCurrentManagedMemory for non-managed memory except it takes an object as argument and return the MemoryArea in which the object is allocated.

initialize
: is a method implemented on classes extending the Mission class and indicate when a mission's memory scope is initial entered. The method is used to initialise event handlers which is then registered in the SCJ API.

In order to enable the analysis to track when the current scope changes, using one of the above function, a call graph must be constructed. Such a graph can be extracted from the information computed by the points-to analysis, since

the set of possible target objects for a dynamic dispatch are already covered by the points-to analysis. Using the call graph it is possible to make an over-approximation of the scope stack of all methods. Based on this it can be determined in which scope stack a method allocates new objects, and thus, using the results from the pointer analysis, it is possible to compare the scope stack of the allocated objects scope with the scope stack of the reference and thus find potential violations of the memory scope rules.

Besides the functions listed above, the functions newArray, newArrayInArea and newInstance can be used to allocate objects in other scopes than the current scope. Our tool does not currently support analysis of these functions, but may be implemented in a way similar to the analysis of handleAsyncEvent. None of the analysed applications made use of these functions and their use is generally discouraged as it may lead to ambiguous analysis results.

It should also be noted that the initialize method can only be used to track when a mission's memory is initially entered. However, there is no API method that can be used to track when the mission's memory is entered again, before the handleAsyncEvent of a periodic event handler is called, as handleAsyncEvent is called from the same level that set up the Safelet [6, Figure 3.3]. This makes it difficult for a static analysis to track when mission memory is entered. In the next section we discuss possible solutions.

4. IMPLEMENTATION

The implementation of the analysis is based on the T.J. Watson Libraries for Analysis (WALA) [1] that provides support for many different kinds of pointer analysis including several context sensitive analyses that allow analysis developers to define and implement specialised contexts (called parameterised context sensitive analyses). Contexts in a pointer analysis are typically based on the call chain. In contrast, we implement specialised contexts consisting of a scope stack (as described in the previous sections). For the implementation of a scope we use a string to represent the MemoryAreaID and a scope stack is implemented using a linked list.

WALA uses an intermediate representation of the Java bytecode to perform the analysis on. The intermediate representation is similar to Java bytecode, but in static single assignment (SSA) form which eliminates the need for a stack and simplifies flow-sensitive analyses. However, the use of SSA makes it slightly more difficult to map the results back to the exact location in the original code. Currently problems are reported using name of fields, methods and classes which we found to be sufficient to locate and correct potential violations of scope rules.

Another feature of WALA is that it performs pointer analysis and call graph construction simultaneously, enabling the construction of a sound call graph, i.e., a call graph that takes dynamic dispatch into account.

The call graph produced by the analysis is built from a number of call graph node objects. Each call graph node represents a method *invocation*, i.e., the execution of the method at runtime. If methods are invoked in different contexts, i.e., different memory scopes, this will give rise to separate call graph node objects. To implement a parameterised context sensitive pointer analysis in WALA, the analysis developer must provide a class implementing an interface called ContextSelector whose implementations can be called at certain points of the analysis while building the call graph in order to change the current analysis context of methods.

```
1  public Context getCalleeTarget(CGNode caller,
2    CallSiteReference site, IMethod callee,
3    InstanceKey[] actualParameters)
4  {
5   ScjContext cc;
6   ...
7   cc = (ScjContext) caller.getContext();
8
9   if (isFuncName(callee, "handleAsyncEvent") &&
10      isSubclassOf(callee.getDeclaringClass(),this.MEHIClass))
11  {
12    cc = new ScjContext(cc,
13                        getClassName(callee),
14                        ScjScopeType.PM);
15  } else if (isFuncName(callee, "enterPrivateMemory") &&
16    isSubclassOf(callee,this.ManagedMemoryIClass))
17  {
18    cc = new ScjContext(cc,
19                        getUniquePMName(),
20                        ScjScopeType.PM);
21  } else if (isFuncName(callee, "getCurrentManagedMemory") &&
22    isSubclassOf(callee,this.ManagedMemoryIClass))
23  {
24    ((ScjContext)caller.getContext()).
25      setLastGCScope(cc.peek());
26  } else if(isFuncName(callee, "executeInArea") &&
27            isSubclassOf(callee,this.MemoryAreaIClass))
28  {
29    cc = new ScjContext(cc, cc.getLastGCScope().getName(),
30                cc.getLastGCScope().getScopeType());
31  } else if (isFuncName(callee, "startMission") &&
32            getClassName(callee).
33              equals("Ljavax/safetycritical/JopSystem"))
34  {
35    cc = new ScjContext(cc,
36                        getClassName(caller),
37                        ScjScopeType.MISSION);
38  }
39
40  return cc;
41 }
```

Figure 2: Implementation of getCalleeTarget for context change.

The WALA implementation of our memory safety analysis was fine-tuned to achieve few false positives. As such parameters were chosen mainly to favour increased precision over performance. Specifically, we set an option for the analysis that causes instances in a method to be distinguished by allocation site rather than simply by type. In order to prevent this option from resulting in state space explosion, another feature was enabled, called SMUSH_MANY, that collapses object instances once more than 25 different object instances are found in a method. By default, similar options are enabled for strings, called SMUSH_STRINGS, and throwables, called SMUSH_THROWABLES. However, through experimentation we found that disabling these leads to fewer false positives with no notable performance degradation.

4.1 Implementing Context Changes

The first step towards a WALA implementation of our memory safety analysis is to define the necessary contexts and, in particular, to define the class implementing the ContextSelector interface which handles context changes.

The ContextSelector interface requires classes implementing it to define the methods getCalleeTarget and getRelevantParameters. In our implementation the most interesting method is getCalleeTarget as it is called for all possible

method invocations in order to determine whether or not to change context for a given method call.

The interesting part is when the method invoked is one of the special context changing API methods discussed in Section 3. Figure 2 shows our implementation of the getCalleeTarget method that we will explain in more detail in the following.

The getCalleeTarget method accepts four arguments: the calling method represented by a call graph node, a representation of the call site, a representation of the called method, and the actual parameters of the call. Based on these arguments the getCalleeTarget method return the context of the called method as a Context object (lines 1–3). For brevity and clarity we have elided the initialisation and handling of the initial call graph nodes, called *synthetic nodes*. The context of these nodes is set to immortal memory such that methods executed in immortal memory are bootstrapped correctly.

Two utility methods getClassName and getUniquePMName are used in getCalleeTarget. The first returns the class name of either the calling or called method. The second returns a unique string used to name an anonymous private memory created by enterPrivateMemory.

For most methods, i.e., all methods that do not change the context, the new context is simply the same as the old context as determined by the calling method (line 7 and 40). The large conditional (lines 9–38) is used to check if the called method is one of the "special" methods and handle it accordingly. Specifically, the name of the called method is compared with the names of the context changing methods and it is further checked that the method is defined in the right class. All the context changing functions, except getCurrentManagedMemory, are handled by assigning a new context with an updated scope stack to the variable cc (e.g., line 12). The scope stack is updated by adding the relevant scope, which comprises a string and a scope type. The string is used to separate different missions, event handlers and private memories.

As a concequence of getCurrentMangedMemory the analysis needs to track the reference returned by this method. Not doing so would mean we would lose all information when executeInArea is used. We noticed that we did not see any application requesting two references before running executeInArea. Therefore we implemented a simple memory reference tracking by expanding the definition of a scope stack with an additional scope:

$$\mathsf{ScopeStack} = \mathsf{Scope}^* \times \mathsf{Scope}$$

This scope is used to annotate scope stacks with the scope returned by getCurrentManagedMemory. If getCurrentManagedMemory is called more than once in a call chain the scope on the scope stack will be assigned UNKNOWN.

In the implementaion getCurrentManagedMemory is handled in (lines 24–25). This is done by calling setLastGCScope(cc.peek()) on the current context/scope stack. This sets the value of a field lastGCScope, on the current context, to the value on the top of the scope stack. If lastGCScope is already assigned a scope it will be assigned a scope with the type set to UNKNOWN to represent an unknown memory. In case of the function executeInArea, the lastGCScope is used to determine which scope the new context is assigned. In some cases this may be an unknown memory (lines 26–30).

4.2 Analysing SCJ

In order to soundly and fully analyse an SCJ application, it is necessary to take the effects of the underlying SCJ implementation and API into account, since important flow information may otherwise be lost. As an example, periodic event handlers (from class PeriodicEventHandler) have to be *registered* in the SCJ implementation, which will then invoke the handleAsyncEvent method on the registered PeriodicEventHandler object at appropriate time intervals.

One possible solution to this problem would be to write stubs for all the relevant classes and methods of the SCJ. However, true to its RTSJ heritage, the SCJ libraries are numerous and sizeable. Instead we initially opted to use an SCJ implementation built directly on top of an RTSJ implementation [12]. However, analysing an application together with this software stack proved to be too time consuming and gave rise to many false positives.

Instead of using the SCJ implementation built on top of an RTSJ implementation we used the JOP/SCJ implementation [9, 10] which turned out to be suffieciently small to provide immediate feedback of analysis results to the programmer on all tested programs.

Using the JOP/SCJ implementation lead to a few minor issues: The first, is that the result of the pointer analysis always contains exactly ten InstanceKey objects, of the declared type String, without an associated context. These are String objects from exceptions allocated in the JOP/SCJ implementation. When checking for violations, we filter these out along with potential violations occurring strictly within the SCJ implementation itself, since the SCJ infrastructure may need to temporarily break the scope rules and is allowed to so. However, to support developers implementing the SCJ specification it is possible to set the variable showPrimordial, on the class Problem, to true to have our analysis report problems in the SCJ implementation as well.

A second issue with the JOP/SCJ implementation was that it did not, at the time, support getCurrentManagedMemory, executeInArea and getMemoryArea. This was solved by making a new branch of the JOP/SCJ implementation and adding support for getCurrentManagedMemory and executeInArea. We have postponed implementing support for getMemoryArea as it requires substantial effort and none of our test programs used this function. The function executeInArea was implemented to only work for objects of the ManagedMemory class.One of the advantages of using the JOP/SCJ implementation is that we can use the JOP/SCJ specific function startMission to handle the issue concerning tracking when a mission's memory is entered (since the initialize method can only be used to detect the first entry into mission memory).

4.3 Finding Potential Safety Violations

The result of the pointer analysis is a list of objects of type PointerKey and associated InstanceKey objects. A PointerKey is an abstract representation of a static field, instance field, or a local variable that stores a reference to an object. An InstanceKey is the abstract representation of objects based on type or, as in this case, allocation site.

The pointer analysis is over-approximating, meaning that in the analysis a PointerKey may point to InstanceKey objects that will never occur during execution of the concrete program. On the other hand, the analysis is sound, meaning that whatever any (concrete) pointer may point to dur-

ing the execution of the concrete program, there will be an abstract representation of it in the analysis result. Such a sound analysis can in general be used to verify safety properties, e.g., to verify that pointers from the immortal memory scope will never point to objects living in mission or private memory during *any* possible execution of the program.

To verify this property we need to ensure that the scope stack ss_{pk} of a PointerKey and the scope stack ss_{ik} of all InstanceKey objects the PointerKey may point to, conforms to the $ss_{ik} \sqsubseteq ss_{pk}$ ordering defined in Section 3.

The main loops for checking the result of the pointer analysis can be seen in Figure 3. The code uses two helper methods getScjContext and report_problems. The first is used to get the context(scope stacks) from PointerKey and InstanceKey objects. The second is used to report problems and extract as much information from the PointerKey and InstanceKey that is avilable. The code shows a loop iterating over all InstanceKey objects. If an InstanceKey does not have an associated context a counter variable i is incremented, filtering away the ten exception strings from the JOP/SCJ implementation. Otherwise all PointerKey objects that may point to the InstanceKey object are iterated over (lines 15–21). While iterating over all the PointerKey objects a test for a mismatch of scope stacks is performed using the less method which implements the $\sqsubseteq$ ordering. If the method returns false a problem based on the information from the involved PointerKey and InstanceKey is added to a set of problems found using the report_problems. The reason for using a set is to eliminate duplicates from the set of problems. Experiments found this to be usefull as problems would otherwise often be reported more than once.

```
1  public static void runAnalysis(PointerAnalysis pa, HeapGraph hg,
2      HashSet<Problem> problems)
3  {
4    Iterator<InstanceKey> ikItr = pa.getInstanceKeys().iterator();
5    int i = 0;
6
7    while( ikItr.hasNext() )
8    {
9      InstanceKey ik = ikItr.next();
10     Iterator<Object> pkIter = hg.getPredNodes(ik);
11
12     if (getScjContext(ik) == null) {
13       i++;
14     } else {
15       while (pkIter.hasNext())
16       {
17         Object pk = pkIter.next();
18
19         if (!getScjContext(ik).less(getScjContext(pk)))
20           report_problems(problems, ik, pk);
21       }
22     }
23   }
24
25   if (i != 10) {
26     util.error("Unexpected␣number:␣"+i+
27       "␣of␣InstanceKey␣objects␣with␣context␣==␣Null");
28   }
29 }
```

Figure 3: Checking the result the pointer analysis for potential problems.

4.4 Example

To demonstrate the advantage of using the memory analysis instead of using annotations an example is presented. The example is taken from Figure 6 in [14]. It shows a program annotated with memory safety annotations (highlighted in the example) and with two parts of code replaced by "...". The annotations are used to define scopes and relation between scopes and classes, references to instances of the ManagedMemory class and methods. The example contains seven lines of annotations. However, we do not find the number of lines of annotations to be significant rather the complexity of learning and using the memory safety annotations. In [14] it is stated that memory safety is the most complex property their analysis checks. A total of 14 rules have to be checked to verify correctness using the annotations. Our analysis means programmers do not have to annotate SCJ programs with memory safety annotations as seen in the example. In fact, the analysis is able to guarantee memory safety for the example, *without* any annotations in the code, in a fully automated way.

While it can be argued that programming with annotations forces the programmer to think more carefully, or at least more explicitly, about memory usage and scope requirements, it can also take up a lot programmer resources and may lead to unfortunate code duplication [14].

In the experiments section we present results from analysing two test cases (pmFFTcpresult and InOutParameter) using a similar pattern although the test cases are more involved.

```
1  @Scope("immortal")
2  @DefineScope(name="MyMission", parent="immortal")
3  class MyMission extends CyclicExecutive {
4    public void initialize() {
5      new MyHandler(...);
6    }
7  }
8
9  @Scope("MyMission")
10 @RunsIn("MyHandler")
11 class MyHandler extends PeriodicEventHandler {
12
13   public void handleEvent() {
14     @DefineScope(name="MyRunnable",parent="MyHandler")
15     MyRunnable r = new MyRunnable();
16     ManagedMemory.getCurrentManagedMemory().
17       enterPrivateMemory(3000, r);
18   }
19 }
20
21 @Scope("MyHandler")
22 @RunsIn("MyRunnable")
23 class MyRunnable implements Runnable {
24   public void run() { ... }
25 }
```

5. EXPERIMENTS

To test the implementation, a number of test cases was used. In Table 1 the results of running the analysis on these test cases are shown. The table lists five test cases and their complexity in the form of lines of code, size of bytecode of SCJ library and application classes respectively, (known) number of illegal assignments, and the number of illegal assignments the analysis found. Due to the lack of real world level 0 and 1 SCJ applications the test cases are based on applications developed as part of two master thesis projects, a modification of one of them, an example developed as part of CJ4ES project activities and a test case developed for another paper with clever reuse of space in a StringBuilder which we expected would result in a false positive. The analysis perform very well and as expected. Below a description of the test cases can be found.

Test case	LOC	Bytecode	IA	Found
scjminepump	1465	239884/18519	0	0
scjminepumplog	1490	239884/20511	1	1
pmFFTcpResult	545	247854/11577	0	0
InOutParameter	155	264949/6285	1	2
scjreprap	1758	242561/27730	4	5

Table 1: This table shows the result of running the analysis on five test cases. LOC is short for Lines of Code. Bytecode size of SCJ library and application. IA is short for Illegal Assignments

scjminepump
The scjminepump is inspired by the classical minepump textbook example. It was implemented as part of a master's thesis project at Aalborg University. The work was later published in [5]. The example has been adapted to use the JOP/SCJ implementation. The example is straight forward and does not pass references between scopes and therefore does not contain any illegal assignments.

scjminepumplog
An extension of the previous test case with logging facilities. For testing purposes this case contains an illegal assignment.

pmFFTcpresult
This test case use nested private memory to encapsulate a Fast Fourier transform library which creates temporary objects that would otherwise have to be reclaimed by a garbage collector. The code demonstrates the usecase of reusing existing libraries in an SCJ application and was developed as part of ongoing activities in the CJ4ES project. The test case uses executeInArea to move the result of the Fast Fourier transform to a higher scope by copying the result data from the inner scope to an object allocated in the outer scope.

InOutParameter
This test case uses private memory for encapsulating code that leaks garbage. The result from the private memory is stored in a StringBuilder allocated in the outer scope. The StringBuilder is initialised with a buffer size of 30. When entering the private memory the length of the StringBuilder is reset to be able to reuse the buffer for output. As our analysis is not data sensitive it cannot guarantee that the buffer size of 30 cannot be exceeded, which would result in allocation of a new buffer in the private memory scope. If a new buffer is allocated, an illegal assignment will occur. Using our memory analysis this gave rise to a false positive as expected. Furthermore, the test case uses a static variable to store a SimplePrintStream object. The object is allocated in mission memory but a reference is stored in immortal memory by use of a static variable. This illegal assignment was not intended by the author and was first discovered by the analysis.

scjreprap
The scjreprap test case is an implementation of the controller software for a reprap 3D printer on top of the JOP/SCJ implementation as part of a master's project [13]. It was unknown whether this test case contained illegal assignments. The test case is relatively large and has an interesting control structure. The analysis found five problems. By examining the code based on the problems reported it was found that four were indeed illegal assignments. The last report is a false positive due to the problem of tracking when a mission starts by examining method invocations. We currently track this by using the JOP/SCJ startMission, unfortunately this means that the method getSequencer on the mission object is called as if it was invoked as if it was executed in mission memory although it in fact is invoked just before entering mission memory.

Overall the results contain few false positive indicating good precision of the memory analysis. The experiments were executed on a Intel Core i7-2620M CPU at 2.70GHz with 8 GB Memory and a solid state disk. The running time of the analysis was timed by using System.currentTimeMillis() before and after the analysis to measure the number of milliseconds spend on running the memory analysis. The experiments were performed three times. All test cases were analysed in around one second. The source code of the experiments can be found in the git repository associated to this paper[1] in the file ScjMemoryScopeAnalysisTest.java. It should be mentioned that the LOC column only lists the application code. The analysis also includes the part of JOP/SCJ implementation that is used in the respective test cases. The entire JOP/SCJ implementation comprises several thousands lines of code. To give an idea of the size of the analysed test cases and the size of the JOP/SCJ implementation analysed the "Bytecode" column list the size in bytes of the JOP/SCJ implementation and the size of the program analysed.

While the fast analysis times of the test cases are not, in themselves, enough to argue that our analysis scales to large applications, the analysis times combined with the documented scalability of WALA in general, does indicate that our approach can scale to cope with large SCJ applications.

5.1 Eliminating Illegal Assignments

Five out of the six known illegal assignments we found were due to the use of static variables. Where developers stored references to objects allocated in either mission or private memories. A general recommendation for reducing false positives therefore is to only use static variables when absolutely necessary. The last, known illegal assignment was due to storing a reference to a logging object, allocated in an inner scope, in an array allocated in an outer scope. This could be avoided by using a number of pre-allocated logging objects.

To eliminate the false positive illegal assignments, the InOutParameter test case could be refactored to avoid using the StringBuilder class for storing the result of computations performed in private memory. Instead an array of characters could be used to store the result. Potentially, a StringBuilder could be used in the private memory and the character sequence could be copied to the character array. The false positive in scjreprap could be avoided by making a small change to the JOP/SCJ implementation or adding a workaround to the analysis. However, we would prefer a more generic way

[1] https://github.com/andreasDalsgaard/privmem

to identify when mission memory is entered across SCJ implementations.

6. RELATED WORK

In [11] a points-to analysis for Java bytecode (actually for the JamaicaVM intermediate language) is developed and applied to finding violations of the RTSJ memory scope rules, as well as for finding simple programmeing bugs such as dereferencing a null pointer. The paper does not discuss scalability of the analysis, however, an example is shown, a "hello world" program, that takes roughly 11 seconds to analyse.

The SCJ specification [6] specify a set of SCJ annotations. The annotations can be divided into three groups: Compliance levels, behavioral restrictions and memory safety.

In [14] an analysis for checking all three groups of SCJ annotations is presented. The implementation is based on the Java Checker framework using the annotations in the application as a type system to restrict the number of permitted applications. An example of 24 thousand lines of code is annotated and verified with their implementation. In total 92 annotations had to be added, of which 31 were memory safety annotations and 61 were related to compliance level. However, the rules for memory safety are far more complex than those for compliance level. This is exemplified by the large number of rules memory safety annotations have to abide by. Given the unfamiliar memory model of SCJ we see it as an additional burden that developers have to annotate the code.

In [7] a related annotation/type system and analysis tool is presented. The annotation system presented is stronger than the annotation system in the SCJ specification. The type system is designed to enable modular composition of software products by, e.g., allowing developers to annotate methods with scope constraints. Although a different annotation system most of the arguments on [14] also apply to this work.

It has previously been shown in [8] that by implementing the check as a hardware runtime check instead of as a software runtime check makes the check around ten times faster. Furthermore, results on two smaller benchmark applications show that this resulted in an average improvement of 18.16% for one of the applications and 0.09% for the other application. This indicate that for some applications it is definitely relevant to eliminate these checks.

A formalisation of the memory model of SCJ have been presented in [4]. The formalisation the Unifying Theories of Programming and may prove an interesting starting point to develop a formal proof of the correctness of the implemented analysis.

7. FUTURE WORK

A formal proof of correctness of the memory analysis should be developed to further increase the confidence in its correctness and facilitate its use for certification purposes. The papers mentioned in the previous section would provide good starting points for such an endeavour. In particular, the paper [4] seems highly relevant, however in the formalisation of private memory for periodic event handlers the scope stack is treated differently from other private memories. Based on experiences with the implementation this could possibly be simplified.

The implementation forms a natural basis for developing an Eclipse plugin enabling highlighting of fields and variables that give rise to potentially illegal assignments. WALA already contains support for mapping results back to the source code. Another way of improving the implementation would be by improving precision. This could be done by modelling the bytecode and using model checking and/or symbolic execution to verify that reported problems can indeed occur. One way of doing this would be by using a Counter-Example Guided Abstraction Refinement approach.

Another way of improving precision of the analysis would be by adding support for manual memory safety annotations. The current implementation does not take any annotations into consideration but could be extended to extract information from, e.g., Compliance Levels or Behavioral Restrictions annotations and thereby reduce ambiguities and false positives. Some way of supporting these annotations would be a useful extension of an Eclipse plugin.

8. SOURCE ACCESS

Our memory analysis implementation is available in open-source, currently hosted at GitHub: `https://github.com/andreasDalsgaard/privmem`.

The memory analysis is based on WALA which should be installed following: `wala.sf.net/wiki/index.php/UserGuide:Getting_Started`

Following the WALA getting started guide will result in WALA being checked out in Eclipse. After finishing the WALA getting started guide the memory analysis should be imported. The analysis is run using the Eclipse Run system using a string similar to:
`-application ${workspace_loc}/app.jar -main /package/Main`
as argument and including wala in the class path.

We used the JOP makefile system to compile jar-files including both the SCJ implementation and application. For instructions on how to do this please see: `http://www.jopwiki.com/Getting_started` Instead of downloading the source of JOP from the main repository a fork that adds support for `getCurrentManagedMemory` and `executeInArea` in the JOP SCJ implementation should be used. This is currently available at: `https://github.com/andreasDalsgaard/jop`

For more details see the README.txt file in the privmem repository.

9. CONCLUSION

In this paper we have shown how a *memory safety analysis* can be used to statically identify potential violations of the SCJ memory model and thereby make it easier for developers to write SCJ applications. This is further strengthened by the fact that the analysis is *fully automated* and does not require the programmer to manually annotate the application.

The analysis is *sound* and can therefore be used to guarantee that an application will never attempt to violate the scoped memory rules. Consequently, the many and potentially expensive runtime checks for memory safety can be dispensed with completely. In addition to improved performance, this is also necessary for safety-critical applications that must meet stringent certification requirements.

We have further discussed, in some detail, our prototype implementation of the analysis, based on WALA, and noted

some of the challenges specific to analysing SCJ programs. We have tested our implementation on a number of benchmarks showing promising results both for performance and scalability, albeit with the caveat that we have not been able to find good, "real-life", and publicly available test applications. We have therefore used a number of "home-grown", but non-trivial, test applications. Still, we believe that static memory safety analysis can be a valuable tool for both development and certification.

10. ACKNOWLEDGMENT

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

11. REFERENCES

[1] T.J. Watson libraries for analysis (WALA). `http://wala.sf.net/`.

[2] P. Amey. Correctness by construction: Better can also be cheaper. *CrossTalk Magazine*, pages 24–28, Mar. 2002.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] A. Cavalcanti, A. Wellings, and J. Woodcock. The safety-critical Java memory model: A formal account. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 246–261. Springer Berlin / Heidelberg, 2011.

[5] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. Wcet analysis of java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 30–39, New York, NY, USA, 2011. ACM.

[6] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java technology specification, public draft, 2011.

[7] K. Nilsen. A type system to assure scope safety within safety-critical java modules. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 97–106, New York, NY, USA, 2006. ACM.

[8] J. R. Rios and M. Schoeberl. Hardware support for safety-critical Java scope checks. In *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, pages 31–38, Shenzhen, China, April 2012. IEEE.

[9] M. Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 47–53, York, UK, September 2011. ACM.

[10] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[11] F. Siebert. Proving the absence of RTSJ related runtime errors through data flow analysis. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 152–161, New York, NY, USA, 2006. ACM Press.

[12] H. Søndergaard. SCJ implementation using RTSJ. `http://it-engineering.dk/HSO/PJ/index.html`.

[13] T. B. Strøm and M. Schoeberl. A desktop 3d printer in safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[14] D. Tang, A. Plsek, and J. Vitek. Static checking of safety critical java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, New York, NY, USA, 2010. ACM.

[15] D. Tang, A. Plsek, J. Vitek, and K. Nilsen. A static memory safety annotation system for safety critical Java. Unpublished paper.