Static Data Flow Analysis for Realtime Java Runtime Error Detection

Fridtjof Siebert

▶ **To cite this version:**

Fridtjof Siebert. Static Data Flow Analysis for Realtime Java Runtime Error Detection. Embedded Real Time Software and Systems (ERTS2008), Jan 2008, Toulouse, France. hal-02270265

HAL Id: hal-02270265
https://hal.science/hal-02270265v1

Submitted on 24 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Static Data Flow Analysis for Realtime Java Runtime Error Detection *

Dr. Fridtjof Siebert
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany
siebert@aicas.com

ABSTRACT

The strict and clear semantics of Java make it an ideal language for static analysis. Nevertheless, the use of program-wide pointer analysis for proving the absence of Java runtime error conditions such as *null* pointer uses or illegal array indices is still not widespread. Current uses of program-wide pointer analysis focus on extracting information for optimisations in compilers. In this case, imprecise analysis results only in less agressive optimisation, which is often tolerable.

Existing implementations of program-wide data flow analysis either lack the required accuracy to prove the absence of large enough numbers of certain errors or cause an explosion in analysis time and space requirements for non-trivial applications. Low accuracy leads to large numbers of "false positives", i.e., code for which the analysis fails to prove that a certain error condition does not occur. An explosion in the analysis effort causes the analysis not to produce any useful results within reasonable time at all.

The approach presented in this paper applies the analysis to create a correctness proof of safety-critical Java applications. This includes the absence of runtime errors (*null* pointer use, division by zero, etc.), the absence of potential deadlocks, the correctness of region-based memory management and the determination of resource constraints (heap and stack use).

Keywords

program analysis, data-flow analysis, Java, Real-Time Specification for Java, context sensitive, pointer analysis, runtime errors

*This work was partially funded by the European Commission's 6th framework program's HIJA project, IST-511718.

1. INTRODUCTION

The enormous success of Java technology is due to the language's many advantages over more traditional languages, such as higher productivity and safety. Even critical applications, as in automotive or aerospace control, can profit from these advantages hidoors, aero [6, 1]. Extensions such as the Real-Time specification for Java (RTSJ) [3] or real-time garbage collection technology [13] make Java implementations fit for the time-critical application domain.

1.1 Scoped Memory in the RTSJ

Language extensions such as the Realtime Specification for Java [3] have made it possible to use Java implementations even though a garbage collector may interrupt the execution of normal Java code in an unpredictable way. Region based memory management [17] using scoped memory and realtime tasks that cannot access the garbage collected heap make it possible to develop code that has predictable timing behaviour and that can be used for hard realtime tasks.

Scoped memory provides an allocation context that is freed when the context is exited. The use of scoped memory, however, requires that certain runtime error conditions are avoided. These new runtime error conditions are scopes that are nested improperly and assignment that may lead to dangling references.

Scoped memory areas may be nested such that entered memory areas form a tree. Each scope has at most one parent scope in this nesting, but it may have several child scopes. Since the same scoped memory area may be used by different threads simultaneously, each thread must follow the same nesting order. Current implementations of the RTSJ check at runtime that the scopes actually form a proper tree and not a cyclic graph. If a cycle would be created by entering a scoped memory area, a $ScopedCycleException$ is thrown at runtime.

Since memory allocated in a scope will be reclaimed after this scope was exited by all threads, the implementation has to ensure that no dangling references to any objects allocated in a scope will exist when this scope is exited. Therefore, the RTSJ does not permit storing a reference to an object allocated in scoped memory into a static variable or into another object that was allocated in heap memory, immortal memory, or in an outer scope. Current implementations of the RTSJ use runtime checks on all pointer assignments to make sure that no assignments that may lead to dangling

references are made. In case an attempt is made to make such an illegal assignment, an $IllegalAssignmentError$ is thrown by the virtual machine.

In addition to $ScopedCycleException$ and $IllegalAssignmentError$, another runtime error that may occur when using scoped memory is, of course, an $OutOfMemoryError$ when more memory is allocated than the scope was declared to hold.

1.2 Pointer Analysis

The application of pointer analysis to object-oriented languages such as Java is a relatively new area of research. There are two main uses of the results of pointer analysis: the results can be used to control optimisations of code manipulating tools such as compilers, or the results may be used for the correctness analysis of an application. In the presented approach, the results are used for correctness analysis, which requires a very accurate analysis to produce useful results.

1.3 Contributions

This paper presents a context-sensitive and flow-sensitive pointer analysis algorithm for an object-oriented environment. The results of this pointer analysis are applied to prove the absence of error conditions. The development of the analysis algorithm was driven by the requirement to reduce the number of false positives in the set of potential errors that is reported without causing a state explosion. The error conditions include standard Java runtime errors such as *null* pointer use, illegal casts and illegal array stores, but also memory related runtime errors in applications using the Real-Time Specification for Java [3]. The analysis results are used for the verification of safety-critical Java applications that are developed according to the profiles defined within the HIJA project [7].

To illustrate the usefulness and scalability of the presented algorithm, it is applied to the real-world Java applications that are part of the SPECjvm98 benchmark suite [14], the time and memory demands for the analysis of these applications are presented as well as the achieved analysis accuracy for runtime error conditions.

2. MOTIVATING EXAMPLE

A typical use of scoped memories is show in figure 1. In This example, a realtime thread runs it its own scoped memory area $s1$. It starts by reading data from several files. A second memory area $s2$ is used to store all the temporary objects required to read those files. However, the data read from the files is allocated in $s2$ and copied directly to $s1$ causing an $IllegalAssignmentError$ at runtime in the call to $Vector.add$ made in line 28.

This kind of programming error is hard to predict since it is caused caused by an apparently harmless pointer store operation. Finding it typically requires complex manual analysis of the code to determine the possible memory areas of the source and the target of the pointer store operation. In this example, the illegal assignment error occurs inside class $Vector$, even though the code of class $Vector$ is correct; the error is in the call at line 28 of the example that passes an argument that cannot be stored into the vector. This example

```
 1: import javax.realtime.*;
 2: import java.io.*;
 3: import java.util.Vector;
 4: public class Test implements Runnable
 5: {
 6:   final static LTMemory s1 = new LTMemory(1000000);
 7:   final static LTMemory s2 = new LTMemory(1000000);
 8:   public static void main(String[] args)
 9:   {
10:     new RealtimeThread(null,null,null,s1,
11:                        null,new Test()).start();
12:   }
13:   public void run()
14:   {
15:     final Vector results = new Vector();
16:     for(int i=0; i<10; i++)
17:       {
18:         final int n = i;
19:         s2.enter(new Runnable() {
20:           public void run()
21:           {
22:             try
23:               {
24:                 RandomAccessFile f = new
25:                   RandomAccessFile("file"+n,"r");
26:                 byte[] a=new byte[(int)f.length()];
27:                 f.readFully(a);
28:                 results.add(a);
29:                 f.close();
30:               }
31:             catch (Throwable t)
32:               {
33:                 t.printStackTrace();
34:               }
35:           }
36:         });
37:       }
38:   }
39: }
```

Figure 1: *Example that causes an IllegalAssignmentError in line 28.*

illustrates that the feature of not causing illegal assignment errors at runtime is not an aspect of a single class or method, but an aspect of a whole application.

To prove the absence of such errors, all assignments have to be analysed in all possible calling contexts they may be used in. The pointer analysis presented in this paper solves this problem. One of the results of the pointer analysis is the value sets of all variables in the application. The representation of these values includes the memory area the values were allocated in, such that assignment errors can be found (see section 4.10).

3. POINTER ANALYSIS

3.1 Background

Significant effort has been undertaken in the area of pointer analysis during the last decades [8]. Pointer analysis typically uses static, program-wide data flow analysis, which is an iterative algorithm that determines an upper bound for the set of values each reference variable in an application may hold. In addition to the set of values for each variable,

the analysis determines a set of invocations, where an invocation is a method call together with context information at the call. The resulting set of invocations is an upper bound for the set of invocations that may be performed during an actual program run.

The analysis starts with an empty set of variable values and the set of invocations containing only the main routine of the analysed application. In each iteration, the set of possible values each variable may hold is joined with the set of values that are assigned to these variables by any method that is in the invocation set. Also, any new invocation that is performed by a method that is in the set of invocations is also added to this set. The iterative analysis continues as long as these two sets grow, stopping when the smallest fix point has been reached, i.e., when the sets of values and invocations remained constant during a complete iteration over all invocations.

Pointer analysis is called context-sensitive when the context of the caller is part of the representation of invocations and values. Context information is usually the call chain that leads to an invocation. However, context may include other information such as the thread that performs the invocation or environmental information such as the current allocation context when region-based memory management is used [17, 16].

An algorithm that is not context-sensitive is called context-insensitive. Context-insensitive pointer analysis identifies invocations and values by the method that is invoked and the source code position that creates a value, respectively. Context-insensitive analysis significantly reduces the analysis complexity, but it yields results that are not accurate enough for the purposes described in this paper: values created in different contexts cannot be distinguished and result in the inability to prove the absence of an error. For example, since the instances of a container that are used by different threads cannot be distinguished, a context-insensitive analysis cannot detect that thread local objects stored into a thread local container are not accessible by another thread that uses a different thread local container instance.

In a context-sensitive analysis, the complete call chain is usually used as context. Since this leads to infinite value sets for recursive routines, the call chain has to be reduced to contain no or only a limited number of cycles. However, even with this restriction, the number of possible call chains typically grows exponentially with the application size making context-sensitive analysis difficult to apply to real world applications.

Pointer analysis can be classified further as flow-sensitive and flow-insensitive. A flow-sensitive analysis respects the order of statements during the analysis of one routine, while a flow-insensitive analysis ignores this order. A flow-sensitive analysis achieves higher accuracy since information available in the control graph (such as a *null* pointer check) can be used to reduce the set of values of a variable. Unlike context-sensitivity, the effect of flow-sensitivity on the performance of the analysis is less critical; the additional overhead is similar to the overhead of routine-wide (global) data-flow analysis that is widely applied in optimising compilers at an acceptable cost.

Object-sensitive points-to analysis for Java was presented by Milanova et. al [10]. In object-sensitive analysis, the context information does not consist of the call chain, but the allocation site of the current object (*this* in Java) is used as context information. This approach brings a significant improvement in accuracy compared to a context-insensitive analysis while the analysis complexity grows less than using a context-sensitive analysis based on the complete call chain.

3.2 Jamaica Data Flow Analysis

For the analysis of the correctness of applications using the Real-Time Specification for Java, a new pointer analysis algorithm using augmented context information has been implemented. This data flow analysis uses the intermediate code representation used by the JamaicaVM [9] static Java compiler generated from Java byte code. Single instructions are more fine-grain than bytecodes. For exmaple, an array element access is split into four independent instructions that check the array for *null*, obtain the array length, check the index value and finally read the array element. This intermediate representation replaces the Java stack and the local variables used in the bytecode by using the static single assignment form [2, 11] instead. The static single assignment form simplifies the local data flow analysis in a single method since the data flow between intermediate commands is explicit.

During the data flow analysis, two sets are determined: the set of invocations (called methods with their context) and the set reference values for each reference field and reference array's elements. The analysis runs iteratively starting with the *main* function of the application [1]. The data flow for all invocations is analysed in each iteration. During the analysis, new values are added to the value sets of fields and array elements and new methods that are found to be called are added to the set of called methods. The algorithm terminates when these sets remained constant during one iteration.

Reference values are identified by the class of the object they represent together with context information. This context information is crucial. When it is too detailed, the number of values explodes and the analysis becomes infeasible, while too little context information results in an analysis that finds more "false positives", i.e., potential errors that actually cannot occur at run-time. The context information of the Jamaica data flow analysis consists of the allocation site, the thread that performed the allocation, the current memory area in use, the set of locks held and a form of object context as proposed by Milanova et. al [10].

To reduce the analysis effort, the analysis is split into major and minor iterations. A major iteration analyses all calls in the set of called methods. For the minor iterations, the called methods are placed into separate age groups. If the analysis of a call causes the call or value set to grow, the

[1] the analysis actually starts with the virtual machine initialisation code that is called before the *main* method. For JamaicaVM, this means that an internal constructor of *Thread* and the static initialiser of class *System* is also initially added to the set of invocations

next iteration is restricted to the methods in the same or younger age groups. This means that repeated analysis of invocations that had now effect on the analysis results are avoided in minor iterations. This analysis also distinguishes object initialisation and object use which significantly improves the analysis accuracy for object oriented applications. Specific properties such as singleton instances and embedded instances are also detected to further improve analysis accuracy.

This is the output of the the analysis run on a HelloWorld application:

```
>jamaica -dfa HelloWorld
Jamaica Builder Tool 2.9 Release 5
DFA ITERATION 1.1.1  CALLS:  490 methods: 183 (2s, 67MB)
DFA ITERATION 1.2.15 CALLS: 1215 methods: 251 (4s, 70MB)
DFA ITERATION 1.3.27 CALLS: 2014 methods: 287 (7s, 73MB)
DFA ITERATION 1.4.3  CALLS: 2206 methods: 323 (8s, 80MB)
DFA ITERATION 1.5.9  CALLS: 2332 methods: 341 (9s, 82MB)
DFA ITERATION 1.6.4  CALLS: 2332 methods: 341 (10s, 86MB)
DFA ITERATION 1.7.1  CALLS: 2332 methods: 341 (11s, 89MB)
DFA ITERATION 1.8.1  CALLS: 2332 methods: 341 (11s, 92MB)
DFA DONE: 11583ms TRACED 302 VALUES, 847 VALUE SETS 2332
INVOCATIONS.
```

Even though a HelloWorld is very small, the analysis has to analyse all the code that is executed during startup of the virtual machine, which leads to a total of 341 methods that in this case. Each line that is printed by the analysis shows the state after one major iteration. The analysis stabilises after 8 major iterations, while at most 27 minor iterations where needed (in major iteration 3).

During the analysis, 2332 invocations where found for 341 methods, i.e., there are 2332 different calling contexts even though there are only 341 different methods that may be called. 302 different reference values where detected and 847 different sets of these values were created to store the values of variables. The total analysis time was less than 12 seconds, a time short enough that enables regular use of the analysis tool, e.g., as part of the normal build process of an application.

4. APPLICATION OF POINTER ANALYSIS RESULTS

Pointer analysis results are used for four purposes: the absence of runtime exceptions such as *null* pointer uses and illegal casts, the correctness of synchronisation, the correctness of the region-based memory management available in the RTSJ and the determination of worst-case memory allocation and worst-case stack use.

4.1 Runtime Error Detection

The runtime errors detected by the presented analysis are *null* pointer dereferencing, type cast and array store errors.

4.2 *null* pointer usage errors

Making *null* a special value for references, that is traced by the data flow analysis, also enables proving the absence of *null* pointer dereferencing. The detection of a potential presence of a *null* pointer use is straightforward: At any point in the program that dereferences a pointer, if the *null* value is part of the value set of the dereferenced variable, there is a potential *null* pointer use. Since all instance and static reference fields in Java are initialised with the *null* value, for a useful *null* pointer use analysis it is essential that the presence of initialisation code that overwrites this *null* value is detected reliably.

4.3 Type cast errors

A type cast in Java performs a runtime check that ensures that the casted reference is assignable to the target type. If this is not the case, a $ClassCastException$ is thrown. The availability of the exact value sets enables the detection of potential class cast exception. When for every cast the set of values that is casted contains only values assignable to the destination type, it is proven that the cast will succeed.

4.4 Array store errors

Java permits the assignment of reference arrays to array variables of a more general element type. To ensure that storing an element in an array does not store a value of an incompatible element type into an array of a more specific element type, a runtime check is performed on every array store. The following code illustrates a code sequence that causes such a runtime check to fail.

```
1:  Object[] a = new Object[10];
2:  String[] b = new String[10];
3:
4:  a[0] = "A String";      // ok
5:  a[1] = new Integer(3);  // ok
6:  a = b;                  // ok
7:  a[2] = "Another String" // ok
8:  a[3] = new Integer(3);  // error!
```

With the availability of complete value sets of all variables, it is possible for each array store to check if all possibly stored values are assignable to all possible target array element types. When this is the case, no array store error may occur at runtime, otherwise the assignment is a potential error.

4.5 Array index errors

To be able to analyse the validity of an array index variable, the analysis follows integer values in a way similar to pointer values. Additionally, for each array that is used by the application, the array length provided to the array allocation expression will be saved with the array reference, such that the length will be available at all uses of exactly this reference to the allocated array.

Furthermore, array accesses frequently occur using an index variable that is incremented repeatedly in a loop. For the analysis to be able to determine a useful maximum value for the index variable, loop continuation conditions of the form $index < array.length$ must be detected by the analysis to deduce that an access of the form $array[index]$ is legal.

With this infrastructure, the analysis is able to prove that the majority of array accesses in a typical application are correct.

4.6 Negative array size errors

With the integer values taking part in the analysis, the runtime error that would occur on the allocation or an array with a negative size can be found trivially: If the array size on any array allocation may be a negative value, such a runtime error may occur.

4.7 Division by zero errors

Divisions by zero in Java cause an arithmetic exception. The proof of the absence of such errors is trivial with the integer value ranges: Only if zero is part of the values of the divisor, a division by zero may occur.

4.8 Correctness of Synchronisation

The correctness of synchronisation in a Java application is a hard problem. Errors such as deadlocks that result from different locking orders between threads require an accurate analysis to be detected statically, while these errors occur only sporadically at runtime and are hard to reproduce.

To be able to verify the correctness of synchronisation, the notion of a thread first needs to be available in the pointer analysis. In Java, a thread is started via a call to method *start* on a new instance of class *Thread* or a subclass of *Thread*. When a new thread is started, the *run* method is called on this thread instance. This thread instance is used by the analysis to identify threads. The context of all invocations contains the thread represented by its thread instance. Whenever a call to *Thread.start* is detected by the analysis, a corresponding invocation of the *run* method using the new thread instance as thread context is added to the set of invocations.

A call to *Thread.start* may spawn an arbitrary number of threads if the target thread instance is not a singleton. To be able to detect thread related errors between several such threads with the same context, a second invocation of the same thread is created using a copy of the value that represents the original thread instance (the same idea is used for thread escape analysis by [18]). The copy is marked with a flag *isTwin* such that results can later be filtered appropriately.

In addition to the thread context, it is required to have information on the set of locks that a thread may hold at any point of the analysis. The context of an invocation is therefore equipped with a set of locks that the thread may hold at the invocation. Any synchronisation performed adds the objects the thread synchronises on to the set of locks that may be held at the current invocation context. Any invocation performed during the synchronised statement sequence will use this set as its synchronisation context.

4.9 Absence of deadlocks

Since the set of locks that are held at each invocation is part of the representation of invocations, the proper nesting of synchronisation can be verified straightforwardly. Potential deadlocks as the one illustrated in Figure 2 can be detected. For this deadlock detection, the order in which locks are entered is recorded for each thread. Then, it is checked if other threads that synchronise on the same values do this in exactly the same order. If other threads may synchronise

```
 1:  new Thread() {
 2:    public void run() {
 3:      synchronized (o1) {
 4:        synchronized (o2) {
 5:      }
 6:    }
 7:  }.start();
 8:  new Thread() {
 9:    public void run() {
10:      synchronized (o2) {
11:        synchronized (o1) {
12:      }
13:    }
14:  }.start();
```

Figure 2: *Potential deadlock between two threads.*

```
1:  class C {
2:    synchronized void m() {
3:      if (check()) {
4:        dosomething();
5:      }
6:    }
7:    synchronized boolean check() {
8:      return state == OK;
9:    }
```

Figure 3: *Typical nested synchronisation pattern*

on objects in a different order, there is a potential deadlock. In this case, the innermost synchronisation that may cause a deadlock is displayed as a potential error.

One important code pattern that occurs frequently in Java code is a synchronised instance method that calls another synchronised instance method as shown in Figure 3. Using the plain pointer analysis results, we will get a false positive potential deadlock reported here since it is not obvious from the pointer analysis results that *check*, when called from *m*, will synchronise on exactly the same object and hence may not cause a deadlock. The analysis therefore has been extended to detect invocations within a synchronised method that invoke another synchronised method on exactly the same target object. In this case, the inner synchronisation will be ignored.

4.10 Correctness of region-based memory management

To be able to verify the correctness of assignments of objects allocated in scoped memory and the absence of scope cycles, the context information for invocations and types is extended with the current allocation context. This allocation context is identified by the corresponding memory area instance. On a call to *enter* of a memory area, the context is set to that memory area for the invocation of *run* method that executes in this area.

4.10.1 Absence of cycles between scopes

Verification of the absence of scope cycles is performed by recording a ordering relation whenever a scoped memory area is entered in a context that uses another scoped memory area as a surrounding allocation context. Whenever a new relation is added to this order, DFA checks that this new relation still respects the single parent rule defined by the RTSJ. When this is not the case, a possible $ScopedCycleException$ is reported.

4.10.2 Verifying assignments

The value that represents an allocated object includes the memory area context of the invocation that allocates the object. This information can then be used to check for all reference stores where an $IllegalAssignmentError$ might occur during runtime. When the assigned reference in the store might be allocated in a memory area that is not equal to or a parent of the target of the store, then a possible $IllegalAssignmentError$ is reported by the analysis.

When applying the analysis to the example from figure 1, the illegal assignment is detected reliably. Figure 4 shows the output of the data flow analysis tool when run on this example. The first line of the output is a summary of the problem: there is a potential illegal assignment to an array at line 530 in class $Vector$ (at bytecode number 48). The following two lines show the assignment that causes the problem: the target of the assignment is an array allocated in class $Vector$ at line 148, while the assigned value is a reference to a byte array allocate in line 28 of Test.java. This array resided in the scoped memory $LTMemory$ created in line 7 of Test.java, which is the scoped memory $s2$. Lines 4 through 10 give the context information for the failing assignment instruction. The context is the method $Vector.addElement$ with the allocation context being the $LTMemory$ created in Test.java in line 7, which is $s2$. Line 5 shows the current thread, which is the RealtimeThread created in Test.java in line 11. Line 6 and 7 show the potential values passed to this routine. Finally, to guide the user to the source of the problem, lines 9 through 17 show an example of call chain that leads to the problem. Here, we can see that the call to $Vector.add$ in line 28 of Test.java is on the call chain.

Since the call chain is not used as context information by the analysis becauuse this would lead to an explosion in the number of invocations, several different call chains may lead to the same invocation. This is why, for each invocation, only one example call chain is shown.

4.11 Worst-case memory usage

The accurate invocation graph that is a result of the presented analysis can be used for automatic analysis of worst-case memory demand of the threads that are part of one application. A traversal of the invocation graph and summing of the memory demand of each invocation results in the total memory demand of each method. For this analysis, however, additional constraint information on maximum recursion depths, maximum loop counts, and maximum sizes of allocated arrays is required. This information is not available from the pointer analysis. Instead, the HIJA project uses additional tools that require annotations in the source code or in separate files to provide these constraints. The correctness of these annotations needs to be verified. The HIJA team hija is investigating the use of formal verification based on the KeY verifier key03 for this.

4.12 Worst-case heap memory use

For worst case heap use, the amount of allocation needs to be summed up recursively starting from the main method of each thread. The sum of the memory allocated within loops needs to be multiplied by the maximum loop count and allocations in recursive methods can be determined by multiplying the maximum recursion depth with the allocation performed within one recursive cycle.

4.13 Worst-case stack use

The maximum stack use can be determined by an algorithm similar to the worst-case heap use, only the stack use does not need to be summed up for different calls that originate in the same method; rather it is sufficient to use the maximum stack use of all called methods. Also, stack use is independent of loop counts, but it strongly depends on maximum recursion depths. This stack use in a recursive method is the product of the stack use in one recursive cycle and the maximum recursion depth.

4.14 First experience

First experience with worst-case memory use determination show that the number of additional constraints that are required is limited such that this information can be provided manually even for moderately complex applications. The analysis itself is fast enough even for larger applications.

5. EXPERIMENTAL RESULTS

5.1 Performance

To measure the performance and accuracy of the presented points-to analysis, the widely applied benchmarks from the SPECjvm98 benchmark suite and a simple "Hello-World" like application were used. Even though a "Hello-World" sounds trivial, due to the complex startup code in Java even such a minimal application requires the analysis of 340 methods. The applications in the SPECjvm98 benchmark suite were not written to be easy to analyze, so they pose a much harder test to the analysis framework than the safety-critical applications for which specific coding guidelines will be developed that ease the analysis.

The performance was measured on a Mobile Intel Pentium 4 - M CPU at 1.8GHz running Linux kernel version 2.4.27 (SuSE 8.2). Figure 5 illustrates the analysis performance: The analysis of these tests took between 5 and 710 seconds and required between 34 and 204MB of memory. The number of distinct reference values that where found during the analysis was between 326 and 2298, the number of invocations was between 1989 and 9353, the number of analyzed Java methods was between 340 and 1755.

Four tests exceeded the set analysis time budget of 30 seconds and had the accuracy of some types reduced. For $jess$, very frequent uses of class $StringBuffer$ lead to a reduction in accuracy for this class, while $raytrace$ and $mtrt$ have two classes $ObjNode$ and $OctNode$ that are used in too many different contexts. Somewhat exceptional is the behavior of $javac$, which has a complex recursive use of many different

```
 1: POTENTIALLY ILLEGAL ASSIGNEMT: to array at java/util/Vector.java:530[48]
 2:      java/lang/Object[][ONEINSTANCE]:5846:5811:1830 (Vector.java:148[18])
 3:  <== byte[]:5994:5814:1830 (Test.java:26[34])
          {IN: javax/realtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])}
 4:  IN METHOD method java/util/Vector.addElement(Ljava/lang/Object;)V
          [MemoryArea: javax/realtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])]
 5:  in thread: javax/realtime/RealtimeThread:1829 (Test.java:10[0])
 6:  (arg[0] : java/util/Vector:5811:1830 (Test.java:15[0]),
 7:   arg[1] : byte[]:5994:5814:1830 (Test.java:26[34])
          {IN: javax/realtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])})
 8:
 9:   1: method java/util/Vector.addElement(Ljava/lang/Object;)V (Vector.java:530)
10:   2: method java/util/Vector.add(Ljava/lang/Object;)Z (Vector.java:680)
11:   3: method Test$1.run()V (Test.java:28)
12:   4: method javax/realtime/MemoryArea.enter(Ljava/lang/Runnable;Z)V (MemoryArea.java:1126)
13:   5: method javax/realtime/MemoryArea.enter(Ljava/lang/Runnable;)V (MemoryArea.java:1092)
14:   6: method javax/realtime/ScopedMemory.enter(Ljava/lang/Runnable;)V (ScopedMemory.java:274)
15:   7: method Test.run()V (Test.java:19)
```

Figure 4: *Detailed output of the data flow analysis when run on the example from figure 1.*

classes for expressions that cause an explosion in the analysis effort. So the accuracy was reduced for these expression classes.

The number of source code positions for which potential errors were found are illustrated in Figure 6. Between 82% and 100% of all pointer uses could be proven not to use a null pointer in these tests. For type casts, for all tests except one more than 85% of the casts could be proven not to fail at runtime, the one exception is *javac* where the reduction of accuracy lead to being able to verify only 40% of the casts. Very good results where achieved for array stores, at most 8% of the array stores could not be proven correct. Finally, about 66% of synchronization could be proven not to cause a deadlock.

In summary, a very high percentage of runtime error conditions could be verified with this analysis. First inspections have shown that many of the remaining false positives can be eliminated by simple coding guidelines or by enhancements of the analysis. An example is the frequent code pattern that calls a method on a field f with an explicit *null* check:

```
if (f != null) {
  f.m();
}
```

This code is dangerous if *null* may be assigned to f by another thread. The current analysis does not record which threads perform assignments to fields, so a potential null pointer use will be reported here. A simple code restructuring using a local variable solves the problem:

```
C local_f = f;
if (local_f != null) {
  local_f.m();
}
```

Now, there is no potential null pointer use. For HIJA, we are collecting these kinds of restructuring to form coding guidelines for safety-critical applications. Of course, the approach of extending the accuracy of the analysis to cover such code without modifications in the application is also being followed.

5.2 Graphical User Interface

To improve the usability of the analysis, a graphical front-end was developed that permits browsing through the tree of source code files while potential errors detected by the analysis will be displayed. An example of this graphical output is shown in Figure 7: Here, a potential dead-lock is shown that was detected by the analysis. The displayed hierarchy starts with the packages of the application, then the classes within a package, the methods in a selected class and the method code itself. Any element that contains none of the selected error kinds will be displayed in green, while elements with potential errors are shown in orange.

6. RELATED WORK

Pointer analysis algorithms have been subject of research for over 25 years [8, 12]. Among the list of open issues that where identified by Michael Hind [8] are scalability, precision of results, addressing client's needs, and the support for Java and object oriented languages. In general, the problem is to obtain sufficient accuracy for practical use without incurring state explosion during execution.

Efficient points-to analysis implementations that scale well to real world applications were originally restricted to con-

Application	Analysis time, sec	Memory demand, MB	# of values	# of invocations	# of methods	# of types with reduced accuracy
check	9	32	505	1989	489	0
compress	12	34	529	3084	506	0
jess	196	204	1748	6016	1005	1
raytrace	103	106	847	7765	691	2
db	18	38	671	3885	592	0
javac	710	188	1454	6986	1755	47
mpegaudio	27	57	2095	6312	692	0
mtrt	102	106	847	7765	691	2
jack	45	70	2298	9353	757	0
hello	5	34	326	1192	340	0

Figure 5: *Experimental results: Analysis performance*

Application	potential null pointers uses / total # of pointer uses	potential illegal casts / total # of ref type casts	potential array store error / total # of array stores	potential deadlocks / total # of synchronizations
check	10 / 1953 (1%)	3 / 49 (7%)	0 / 45 (0%)	0 / 48 (0%)
compress	33 / 1813 (2%)	4 / 54 (8%)	1 / 43 (3%)	5 / 43 (12%)
jess	552 / 5265 (11%)	16 / 120 (14%)	1 / 105 (1%)	5 / 60 (9%)
raytrace	452 / 3398 (14%)	5 / 58 (9%)	2 / 57 (4%)	20 / 60 (34%)
db	102 / 2370 (5%)	5 / 73 (7%)	1 / 39 (3%)	5 / 56 (9%)
javac	1726 / 9884 (18%)	213 / 360 (60%)	18 / 457 (4%)	17 / 87 (20%)
mpegaudio	112 / 9605 (2%)	9 / 60 (15%)	1 / 944 (1%)	5 / 42 (12%)
mtrt	452 / 3400 (14%)	5 / 58 (9%)	2 / 59 (4%)	20 / 60 (34%)
jack	286 / 5103 (6%)	8 / 141 (6%)	1 / 99 (1%)	5 / 45 (12%)
hello	0 / 1012 (0%)	0 / 41 (0%)	0 / 21 (0%)	0 / 35 (0%)

Figure 6: *Experimental results: Numbers of potential runtime error conditions found. The numbers are given in number of source code positions, i.e., without context information. The actual contexts that lead to a potential error are nevertheless available in a detailed output of the analysis tool*

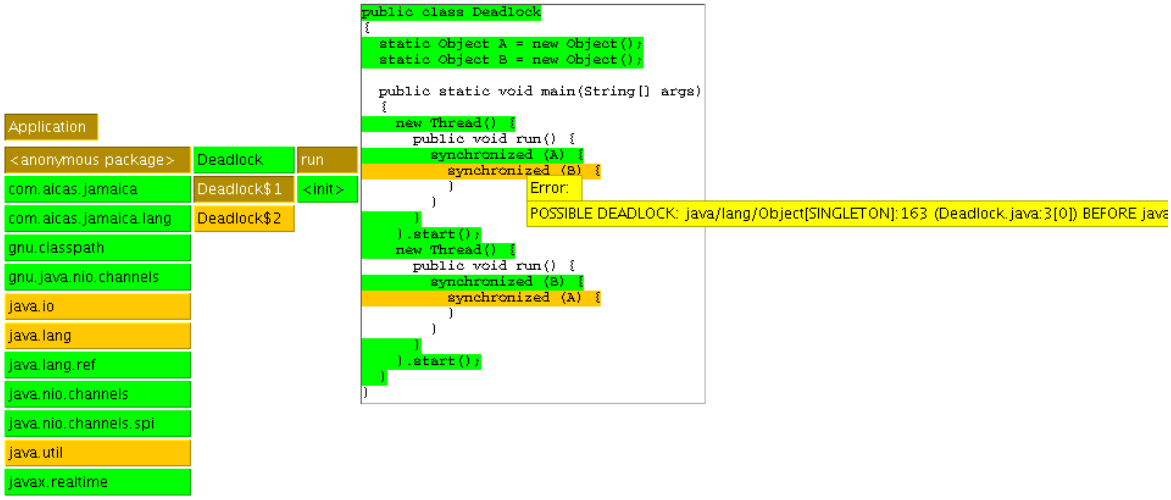

Figure 7: *Graphical representation of DFA results.*

text-insensitive and flow-insensitive analysis. The first context-insensitive and scalable implementation was provided by Steensgard [15]. More recently, context-sensitive points-to analysis that scales well to large C applications has been presented by Fähndrich et al [5, 4]; however, these approaches are either flow-insensitive or unification based. A unification based algorithm regards assignments as bidirectional such that the source and target variable of an assignment has the same value set, which leads to significantly reduced precision compared to the inclusion based approach that is presented here. Furthermore, these earlier approaches differ from the presented approach in that they are not field-independent, whereas the approach presented here maintains separate value sets for each field.

7. CONCLUSION

The presented pointer analysis can prove the absence of pointer and integer related errors such as *null* pointer use or array index errors, but also errors related to the use of region based memory management using the mechanisms available in the Real-Time Specification for Java. The analysis serves as a basis for worst-case memory use analysis enabling static analysis to avoid resource related runtime errors. This is of particular interest for safety-critical code that needs to be certified.

It has been shown that this approach scales well to medium size real-world applications and can prove the correctness of a high percentage of statements that are potential sources of runtime errors.

Future work needs to focus on the development of coding guidelines that ensure a more accurate analysis and that avoid certain obvious problems. Also, enhancements in the analysis accuracy may be achieved by recording additional information such as the set of threads that modify a field.

8. REFERENCES

[1] AERO-VM, the hard realtime virtual machine for onboard space systems. www.aero-vm.com, 2003.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[3] G. Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.

[4] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 175–198, London, UK, 2000. Springer-Verlag.

[5] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 253–263, New York, NY, USA, 2000. ACM Press.

[6] HIDOORS, High Integrity Object-Oriented Realtime Systems. www.hidoors.org, 2002-2004.

[7] HIJA, High-Integrity Java. www.hija.info, 2004-2006.

[8] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001.

[9] Jamaica virtual machine. www.aicas.com/jamaica, 1999-2008.

[10] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[11] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[12] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis.*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[13] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 94–103, 2007.

[14] *SPEC JVM89 Benchmarks*. http://www.specbench.org/osg/jvm98/.

[15] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[16] M. Tofte. A brief introduction to Regions. In R. Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 186–195, Vancouver, Oct. 1998. ACM Press.

[17] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, Feb. 1997. An earlier version of this was presented at POPL94.

[18] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.