Supporting WCET Analysis with Data-Flow Analysis of Java Bytecode


Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology
Treitstraße 3/182-1
A-1040 Wien, Austria
wpuffits@mail.tuwien.ac.at

Abstract

Worst case time execution time (WCET) analysis is crucial for developing hard real-time systems. Many WCET analysis tools rely on manual annotations, which are however not an ideal solution, because they have to be maintained and checked for correctness. Furthermore, a WCET analysis tool for object oriented languages requires knowledge about the receivers of invoked methods in order to minimize the pessimism for virtual function calls.

In this paper, a data-flow analysis framework is presented, which was developed to aid WCET analysis. Currently, a receiver type analysis and a loop bound analysis are implemented. However, the framework is not limited to these analyses and may be extended as needed. The framework is tailored to the needs of the Java Optimized Processor, and the implemented analyses model its behavior accurately. Nevertheless, the framework and the analyses should be adaptable for other platforms with reasonable effort.

This paper also presents results that show the feasibility of our approach. Furthermore, the results provide an insight into programming idioms that are difficult (or impossible) for automatic program analysis.

1 Introduction

In hard real-time systems, missing a task’s deadline can lead to critical system failures, potentially putting human lives at risk. Worst case execution time (WCET) analysis computes a safe bound for the execution time of tasks. These bounds, together with schedulability analysis, can be used to prove that all deadlines can be met. Besides being safe such a bound should also be tight, so the platform can be dimensioned as close to the real requirements as possible and costs are minimized.

On the one hand, WCET analysis has to take into account the performance characteristics of the underlying hardware. On the other hand, WCET analysis has to reason about the behavior of programs, especially with regard to loop bounds. The information about the constraints for the program flow are also referred to as flow facts. Many approaches use manual annotations to communicate flow facts to the analysis tool. A survey on annotation languages can be found in [11].

A manual annotation of flow facts is not an ideal solution. The respective information is specified in two places: the program code and the annotation itself. Inevitably, this can lead to inconsistencies,
especially if the logic of a program is updated. It is not trivial to prove that an annotation is correct
(Hunt et al. [10] try to achieve this by using the Java Modeling Language). In some cases, it would
also be necessary to annotate library code (e. g., String.equals()), which is not always possible.

While data-flow analysis (DFA) cannot solve all these problems, it can remove the need for a
number of annotations. In general, DFA cannot find all loop bounds. Some knowledge of loop
bounds resides outside the program itself, e. g., the memory size. Some knowledge is lost due to the
abstractions an analysis makes (e. g., modeling the possible values of a variable as interval); such
abstractions are however required to keep the computational effort of the analysis feasible. Adhering
to a certain programming style can help the analysis to find loop bounds in such cases. Consider the
following example:

```java
while (cond) {
    foo ();
    cond = ...
}
```

With annotations, this code snippet would look like as follows, given that the loop executes at most
100 times:

```java
while (cond) { // @WCALoop=100
    foo ();
    cond = ...
}
```

Transforming the code snippet to a more analysis friendly style would make it look like this:

```java
int bound = 0;
while (cond && bound++ < 100) {
    foo ();
    cond = ...
}
```

While the annotated and the analysis-friendly versions look fairly similar, the latter version is
superior in one regard: the bound is compiled into the code and the WCET analysis sees the same
bound as the final program. Furthermore, call string dependent annotations are quite cumbersome.
Context sensitive DFA propagates call string dependent constants naturally to the receiver.

We developed a lightweight framework that implements a worklist based analysis kernel similar
to the one presented in [13] and operates on Java bytecodes. The analyses themselves resemble
traditional bytecode interpreters, but operate on abstract states, e. g., intervals of integer values. Two
analyses to support the WCET analysis were implemented: a receiver type analysis and a loop
bound analysis. Our goal has been to implement algorithms useful for WCET analysis; the core
techniques for these analyses were already existent, but the implementations are slightly different
than the approaches we are aware of.

This paper is an extension of the description of the DFA framework in [16], where space and
scope made it necessary to leave out a number of details.

The following section gives a brief introduction of DFA. The framework and the implemented
analyses are described in Section 3. Section 4 presents an evaluation of the framework and our
findings w. r. t. analysis-friendly programming idioms. Related work is examined in Section 5. The
paper is concluded by Section 6, which also provides an outlook on future work.
Data-Flow Analysis

This section gives a very short overview of the foundations of DFA. A thorough introduction to this topic can be found in [13], which provided the basis for this section.

For data-flow analysis, programs are viewed as graphs, where nodes represent statements and edges represent the control flow. The statements generate and update information, which then flows along the edges to other statements. A DFA specifies what information is initially known, how statements update the information, and how information is joined when multiple edges enter a node. Combining this specification with an actual program yields an equation system. When analyzing a program, this equation system has to be solved.

The underlying algorithm for solving a DFA equation system is a worklist algorithm. Such an algorithm iteratively solves the data-flow equations. It iterates over the statements of the program to be analyzed and propagates information along the program flow, until the computed information converges. The requirement of convergence entails that analyses must compute information that is ordered in some way, and the corresponding lattice has a finite height. This means that the top element of the lattice can be reached from the bottom element with a finite number of steps.

As an example for such a lattice consider an analysis that determines whether an integer variable has a constant value. The corresponding lattice has the elements \( \{\bot, \top\} \cup \mathbb{Z} \), where \( \bot \) means that no information is available, any value in \( \mathbb{Z} \) indicates a constant value and \( \top \) means that the variable may not be a constant. The elements of this lattice \( \mathbb{Z}^\top_{\bot} \) are ordered such that \( \forall x \in \mathbb{Z}^\top_{\bot} : \bot \sqsubseteq x, \forall x \in \mathbb{Z}^\top_{\bot} : x \sqsubseteq \top \) and \( \forall x, y \in \mathbb{Z} : x \sqsubseteq y \iff x = y \). This means that \( \bot \) is “less than” any constant value and \( \top \) is “greater than” any constant value. Constant values are only comparable within the lattice if their values are equal. The analysis may only compute information that is “greater than or equal” to the already computed information, and must therefore either reach a fixed point or the top element after a finite number of steps. Figure 1 shows a graphical representation of the lattice \( \mathbb{Z}^\top_{\bot} \).

The Analysis Framework

We implemented an analysis framework that operates directly on Java bytecodes and takes the characteristics of the Java Optimized Processor (JOP) [15] into account. Unlike machine code, Java bytecode contains symbolic information about types, field names etc. As JOP executes Java bytecode with only minor modifications, an analysis at this level is able to model the precise behavior

---

1This is a simplification of the definitions presented in [13], but is accurate enough for the scope of this paper.
2Most notably, the binaries for JOP are prelinked and symbolic information is stripped. However, this does not change the semantics of the program.
of JOP, while still being able to provide symbolic feedback. Having symbolic information at hand also allows easy interfacing with a WCET analysis tool.

The framework builds on other classes within the tool chain for JOP, e.g., to build the transitive hull of classes. As these classes use BCEL [2] for bytecode manipulation, the DFA framework also uses BCEL classes where appropriate. We chose not to use an analysis framework like PAG [1] or soot [20], because the overheads for switching from the existing infrastructure to such a framework did not seem to be justified.

3.1 Analysis Interface

Listing 1 shows the interface that all analyses have to implement. An Analysis uses a ContextMap to represent the current information at a program flow edge. A ContextMap is a simple Map, augmented with context information like the call string and the current stack depth. The context contains more information than necessary to distinguish analysis contexts; information like the stack depth is part of the machine context, i.e., the context information a real JVM would have to maintain for proper execution. The parameters K and V specify the data structures for the information to be computed.

The initial() and bottom() methods return the initial information for the entry node and all other nodes, respectively. The initialize() method is used to set up the Analysis object during the bootstrap process, which is described in Section 3.2.

The method transfer() is the core of the analysis, and specifies what information each bytecode generates and how it transforms incoming information. The current statement is represented by stmt; edge is the edge to which the information flows. Knowledge about the outgoing edge is important for conditional statements, where the information that is passed depends on whether the condition evaluates to true or false. The information that flows into the statement is passed through the parameter input. The parameters interpreter and state are necessary for our method invocation implementation, which is described in Section 3.3.

The comparison of two result sets (the \(\subseteq\) relation) is implemented through the compare() method. Two result sets are merged with the join() method. The methods getResult() and printResult() are used to obtain results in a form that can be used for further processing.

3.2 Bootstrapping

Although being the starting point of program execution, the main() method is not an appropriate starting point for the analysis. A JVM initializes various data structures before the main() method is invoked. In the case of JOP, the class initializers are also executed before the start of the program. These initializations have to be reflected in the analysis.

In the tool described in this paper, a dummy \(<prologue>\) method is automatically generated. It invokes the methods for the JVM internal initializations, the class initializer methods, and the main() method. The \(<prologue>\) method can be regarded as counterpart of JOP’s Startup.boot() method.

3.3 Method Invocation

Virtual function calls are obviously a key feature of an object oriented language like Java. Therefore, the method to be invoked is not known a priori at the call site. This in turn means that the invocation of a method cannot be hardcoded into the program flow representation of a DFA.
Our approach to this issue is simple: the transfer function handles method invocations by recursively analyzing them. For a method invocation, the transfer function therefore has to: (a) set up the information to be passed on, (b) decide which methods may be invoked, (c) analyze the possibly invoked methods, and (d) join the returned information.

To reduce the possibilities for step (b) of this sequence, the first analysis to be run is the receiver type analysis, which is described in detail in Section 3.5. It determines on the fly which virtual methods may be invoked and then recursively analyzes the appropriate methods. The call graph is therefore built up dynamically in this analysis. Analyses that follow the receiver type analysis can rely on the already computed call graph.

A drawback of this solution is that exceptions cannot be handled easily, because an exception may trigger a transfer of control across methods. The proper handling of exceptions is therefore still an open issue for our framework.

### 3.4 Worklist Algorithm

Listing 2 shows a simplified version of our worklist algorithm. In lines 4 to 6, the worklist is initialized and filled with edges to the successors of the entry node. In lines 8 to 16, the result map is initialized. The entry node is mapped to the initial() value for the analysis, all other nodes to a bottom() value. The kernel of the algorithm is displayed in lines 18 to 27. An edge is taken from the worklist, and the transfer() function of the analysis is applied to the statement at the edge’s tail. The resulting information t is compared to the result already computed for the head statement. If the information has changed (or, technically speaking, is not ordered appropriately), the results are merged with the join() function and stored in the result map. Edges to the successors of the head statement are also added to the worklist. The algorithm terminates when the result set has converged; depending on the analysis, the result represents a maximal or minimal fixed point.
Listing 2: Worklist algorithm

```java
interpret(Statements stmts, Flow flow, Statement entry, Analysis analysis) {

  // initialize worklist
  worklist = new List();
  worklist.add(flow.outEdges(entry));

  // initialize result
  result = new Map();
  for (s in stmts) {
    if (s != entry) {
      result.put(s, analysis.bottom());
    } else {
      result.put(s, analysis.initial());
    }
  }

  // interate until convergence
  while (!worklist.isEmpty()) {
    (head, tail) = worklist.first();
    worklist = worklist.rest();
    t = analysis.transfer(tail);
    if (!analysis.compare(t, result.get(head))) {
      result.put(head, analysis.join(result.get(head), t));
      worklist.add(flow.outEdges(head));
    }
  }
}
```

3.5 Receiver Types

Receiver types determine the version of the method a virtual function call actually invokes. As this depends on the type of an object, the analysis propagates the possible types for all fields and local variables through the potential execution paths.

The bytecode `new`, for example, creates a mapping from the top-of-stack element to the type of the created object. A `putfield` instruction takes the type of the appropriate stack entry, adds the field name and creates a mapping from this field location to the type of the top stack entry. When allocating arrays, the allocation site is added to the type information. This helps to distinguish arrays that are instantiated in different places and thus makes the computed information more precise.

3.5.1 Example

The receiver type analysis maps Locations to sets of Types. A Location represents a stack entry or a location on the heap. A stack entry is modeled as integer value \( s \in \mathbb{N} \), “pointing” to the according stack entry. A heap location consists of a Type and a field name; in case of an array type, all array fields are merged to a single field. A Type is a string that contains the fully qualified class name and for arrays and constant strings also the position at which the instance of this Type was created.
Listing 3: Receiver type analysis code example

```java
public void foo();
...
13: new class Bar();
16: dup;
17: invokespecial Method Bar.<init>:()V;
20: putstatic Field Baz.baz:LBar;
...
```

Adding the allocation site allocation allows distinguishing array and constant string instances in later analyses. It would also be possible to add the site of allocation for regular classes, but the analysis then quickly becomes infeasibly expensive in terms of both memory consumption and computation time.

Consider the bytecode fragment in Listing 3. The bytecode at position 13 creates a new mapping for the top stack entry, e.g., stack slot 3, resulting in the mapping `{..., stack[3] → Bar, ...}`. The subsequent `dup` bytecode yields the following mapping: `{..., stack[3] → Bar, stack[4] → Bar, ...}`. The invocation of the constructor pops off the top element and therefore results in `{..., stack[3] → Bar, ...}`. Finally, `putstatic` stores the top stack entry to the static field `Baz.baz`. The result is the mapping `{..., Baz.baz → Bar, ...}`. The analysis conceptually does the same as an actual JVM, but uses symbolic locations instead of actual references to objects.

When invoking a virtual method, the analysis looks up all mappings for the appropriate stack entry, and accordingly considers all potentially called methods. Moreover, it records the receiver types for field accesses, such that subsequent analyses can reuse this information.

3.5.2 Discussion

Our analysis builds up the call graph step by step, and remembers information for each bytecode. Therefore, it is similar to k-CFA ("kth-order control-flow analysis") [17, 18]. As our current implementation does not distinguish different call strings, it can be classified as 0-CFA. In our experiments (see Section 4), the analysis turned out to be rather complex for some programs. Therefore, we abstained from adding call strings, which would have most likely made the analysis too expensive for these programs.

The analysis is flow sensitive and computes information for individual bytecodes. Therefore, it is potentially more precise than other analyses, but it also requires more effort than the techniques presented in the study of Bacon [3] or the approach in [19]. Implementing one of these techniques would be probably necessary to handle large programs.

Part of the complexity for our analysis also comes from the fact that it can handle multiple threads. In the case of multiple threads, the analysis has to analyze all threads until the information converges. Therefore, each thread has to be analyzed multiple times to take the effects of all other threads into account. For proper operation in case of multiple threads, the analysis also has to fall back to a more conservative transfer function for field updates. The more conservative transfer function increases the amount of data to be handled by the analysis and thus makes the analysis more complex.

The receiver type analysis is the first analysis to be run, so subsequent analyses can rely on the information it computes.
3.6 Loop Bounds

To determine the loop bounds in a program, we compute the possible values of integer variables. As it would be infeasible to do this precisely, the analysis operates on intervals of integer values, which is a natural abstraction for such an analysis. Integer operations operate on these intervals, pushing a constant $c$ onto the stack results in an interval $[c, c]$. If an input to an integer operation is unknown or if an operation cannot be reasonably defined on intervals, it results in the interval $[−\infty, \infty]$.

To keep the analysis reasonably precise while still being efficient, we also remember the constraints that appear when a conditional jump depends on the value of some variable. Consider the simple loop shown in Listing 4. Inside the loop body, $i$ is known to be smaller than 50, while it is greater than or equal to 50 when the loop is exited. This information can be used to widen the interval for $i$ to $[0, 49]$ inside the loop body.

The third “ingredient” for the analysis is an interval for the possible increments of a variable. In the example above, $i$ is always incremented by 1, so the appropriate interval is $[1, 1]$. Other operations than incrementing and adding or subtracting known intervals are not supported and result in an increment interval $[−\infty, \infty]$.

For the conditional branch in the header of the loop, the analysis computes the following information:

- $i$ has a value within the interval $[0, 50]$ when the condition is evaluated.
- $i$ is within the interval $[0, 49]$ when jumping to the loop body, and therefore within the interval $[50, 50]$ when exiting the loop.
- $i$ is monotonically increased by a value in $[1, 1]$.

From this information, it can be deduced that the loop body is executed at most 50 times.

3.6.1 Formal Definitions

We define an Interval as follows:

$$\text{Interval} = \{\bot\} \cup \{\{z_1, z_2\}|z_1 \leq z_2, z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}\}$$

With $\sup(int)$ being the upper bound (“supremum”) of an Interval $int$ and $\inf(int)$ being the lower bound (“infimum”), we can define the $\sqsubseteq$ relation on two Interval values $int_1$ and $int_2$ as follows:

$$int_1 \sqsubseteq int_2 \iff \inf(int_1) \geq \inf(int_2) \land \sup(int_1) \leq \sup(int_2)$$

The loop bound analysis operates on tuples of values; the elements of such a tuple $\langle\text{assigned},\text{constrained},\text{increment},\text{source},\text{defscope},\text{softinc},\text{cnt}\rangle$ are described in Table 1. The
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assigned</td>
<td>Interval</td>
<td>The <em>assigned</em> interval contains all values a variable may hold. The bottom element ⊥ represents that no information is available yet. The top element is the interval ([-∞, ∞]).</td>
</tr>
<tr>
<td>constrained</td>
<td>Interval</td>
<td>An interval that represents the known constraints for a variable. A condition like if (x &lt; 10) yields the interval ([-∞, 9]) for the body of the if-statement.</td>
</tr>
<tr>
<td>increment</td>
<td>Interval</td>
<td>An interval that represents all values by which a variable may be incremented. After adding a known interval to a variable, the <em>increment</em> interval contains at least this interval. For example, adding an interval ([2, 2]) to a value with an <em>increment</em> interval ([1, 1]) yields the <em>increment</em> interval ([1, 2]). Other operations than adding or subtracting a known interval result in the <em>increment</em> interval ([-∞, ∞]).</td>
</tr>
<tr>
<td>source</td>
<td>Location</td>
<td>The <em>source</em> location indicates that a value is a copy of some other location. This information is necessary to propagate the constraints generated by a condition back to the actual variable. In Java bytecode, conditions evaluate the top stack entry, not directly a variable.</td>
</tr>
<tr>
<td>defscope</td>
<td>N</td>
<td>Conditional statements are assigned unique scope numbers, in the order as they appear to the analysis. When a variable is assigned a value that is not based on the previous value of the variable, the <em>defscope</em> is set to the scope number of the last conditional statement. If the <em>defscope</em> of a loop variable is greater than or equal to the loop condition scope, no valid bound can be computed. This catches the case where a loop variable is redefined inside the loop body.</td>
</tr>
<tr>
<td>softinc</td>
<td>{0, 1}</td>
<td>Conditional statements set this flag for the variable they depend on. The transfer and join functions take the flag into account, such that a loop bound can only be computed if the variable is incremented in all paths in a loop body.</td>
</tr>
<tr>
<td>cnt</td>
<td>N</td>
<td>The <em>cnt</em> value is incremented every time the <em>assigned</em> interval changes. After reaching a threshold, the <em>assigned</em> interval is widened to ([-∞, ∞]); reaching a second threshold widens the <em>constrained</em> interval to ([-∞, ∞]). By doing so, it is ensured that the information eventually converges.</td>
</tr>
</tbody>
</table>

Table 1: Information for loop bound analysis

The comparison operator \(\sqsubseteq\) operates only on the *assigned*, *constrained*, *increment*, and *softinc* elements. For two tuples \(x_1 = \langle a_1, c_1, i_1, s_1, o_1, n_1 \rangle\) and \(y_1 = \langle a_2, c_2, i_2, s_2, o_2, n_2 \rangle\), we define \(\sqsubseteq\) as pointwise application of the \(\sqsubseteq\) relation for the intervals and equality for the *softinc* flag:

\[
x_1 \sqsubseteq x_2 \Leftrightarrow a_1 \sqsubseteq a_2 \land c_1 \sqsubseteq c_2 \land i_1 \sqsubseteq i_2 \land o_1 = o_2
\]

Joining two intervals \(int_1\) and \(int_2\) is straightforward defined as

\[
int_1 \sqcup int_2 := \left[ \min(\inf(int_1), \inf(int_2)), \max(\sup(int_1), \sup(int_2)) \right]
\]

9
Joining two tuples is more complex, because constraining and widening of intervals takes place implicitly. Listing 5 shows the code for the joining function. In line 5, a copy of the original tuple is created. In lines 7 to 18, it is checked if \( \text{cnt} \) has reached a threshold, and the assigned and constrained intervals are joined or replaced by an interval \([ -\infty, \infty ]\). In line 20, the assigned interval is constrained with the constrained interval, which means that the intersection of the two intervals is computed. The \( \text{widen()} \) method in line 22 widens the assigned interval, such that its bounds are extended to the finite bounds of the constrained interval. As an example, consider the case that \( \text{assigned} = [0, 0] \) and \( \text{constrained} = [ -\infty, 9 ] \). \( \text{assigned}.\text{widen(} \text{constrained} \) will then result in \( \text{assigned} = [0, 9] \). This widening operation has the effect that the \( \text{assigned} \) interval quickly converges and thus speeds up the analysis. In lines 24 to 34, the increment intervals are merged; the code is different from joining the other intervals, because it might be null in both tuples when the tuples are joined. Furthermore, the value of the \( \text{softinc} \) flag is taken into account. If this flag is set, the increment interval is joined with the interval \([0, 0]\). In effect, this prevents the computation of a loop bound if a value is not incremented in all paths of a loop body. The \( \text{defscope} \) is updated with the maximum of the \( \text{defscopes} \) of the two tuples. The \( \text{cnt} \) value is finally updated in lines 38 to 40 if the original \( \text{assigned} \) interval does not match the new \( \text{assigned} \) interval.

After the actual analysis, loop bounds can be computed. For conditional bytecodes, information for the true and the false edge is recorded during the analysis. If the \( \text{assigned} \) interval at an edge is finite and non-empty, and the \( \text{increment} \) interval is not \( \perp \) and does not contain zero, and \( \text{defscope} \) is less than the scope number of the condition, the bound for an edge can be computed as

\[
\text{bound} = \left\lceil \frac{\sup(\text{assigned}) - \inf(\text{assigned}) + 1}{\min(\mid \inf(\text{increment})\mid, \mid \sup(\text{increment})\mid)} \right\rceil
\]

If both edge bounds are valid, the bound for a loop that is controlled by the respective conditional bytecode is the maximum of the two edge bounds. In any other case, no valid loop bound can be computed.

3.6.2 Discussion

The loop bound analysis is inter-procedural and therefore propagates information between different methods. As it also takes the length of arrays and strings into account, it can usually bound loops that iterate through the elements of these data types.

The analysis is agnostic of loops and considers all conditional bytecodes. On the one hand, this approach allows finding value ranges for usual \( \text{if} \)-statements and can therefore be used to find infeasible paths. On the other hand, the analysis cannot distinguish operations inside a loop from operations outside a loop. Therefore, the analysis cannot bound some loops where the loop variable is modified outside the loop body.

Currently, the analysis does not handle multi-threaded programs properly if a loop bound depends on a value or variable that is shared between threads. For the test cases described in the following section, this did not turn out to be a problem. However, future work will have to extend the analysis accordingly.

4 Evaluation

For the evaluation, we use two groups of benchmarks, as shown in Table 2. The first group comprises five benchmarks, three of which are derived from industrial applications (Lift, Kfl, and UdpIp). The
Listing 5: Joining two loop bound analysis tuples

```java
// in class ValueMapping
public void join(ValueMapping val) {
    if (val != null) {
        final Interval old = new Interval(assigned);

        // merge assigned values
        if (cnt > ASSIGN_LIMIT) {
            assigned = new Interval();
        } else {
            assigned.join(val.assigned);
        }

        // merge constraints
        if (cnt > CONSTRAINT_LIMIT) {
            constrained = new Interval();
        } else {
            constrained.join(val.constrained);
        }

        // apply new constraints
        assigned.constrain(constrained);
        // widen if possible
        assigned.widen(constrained);

        // merge increments
        if (increment == null) {
            increment = val.increment;
            softinc = val.softinc;
        } else if (val.increment != null) {
            increment.join(val.increment);
            if (softinc || val.softinc) {
                increment.join(new Interval(0, 0));
                softinc = true;
            }
        }

        defscope = Math.max(defscope, val.defscope);
        if (!old.equals(assigned)) {
            cnt++;
        }
    }
}
```

second group are benchmarks provided by the Mälardalen Real-Time Research Center [12] and ported to Java by Trevor Harmon [7]. As the benchmarks are hardly object oriented, the results for the receiver type analysis are not representative and therefore not further evaluated in this section.

Table 2 shows, besides a description, also the total lines of code (LOC) for the benchmarks, the number of loops in the application code, the percentage of detected loop bounds and the time for
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Loops</th>
<th>Bounds (%)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>CRC calculation for short messages</td>
<td>92</td>
<td>1</td>
<td>100</td>
<td>2.2</td>
</tr>
<tr>
<td>SVM</td>
<td>Machine learning algorithm</td>
<td>3079</td>
<td>7</td>
<td>43</td>
<td>210.2</td>
</tr>
<tr>
<td>Lift</td>
<td>Lift controller</td>
<td>675</td>
<td>6</td>
<td>67</td>
<td>5.2</td>
</tr>
<tr>
<td>Kfl</td>
<td><em>Kippfahrleitung</em> application</td>
<td>1614</td>
<td>18</td>
<td>67</td>
<td>12.6</td>
</tr>
<tr>
<td>UdpIp</td>
<td>UDP/IP benchmark</td>
<td>1412</td>
<td>12</td>
<td>50</td>
<td>7.5</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>Binary search program</td>
<td>78</td>
<td>1</td>
<td>0</td>
<td>2.2</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>Bubblesort program</td>
<td>63</td>
<td>3</td>
<td>100</td>
<td>1.8</td>
</tr>
<tr>
<td>Crc</td>
<td>CRC calculation</td>
<td>154</td>
<td>3</td>
<td>67</td>
<td>5.4</td>
</tr>
<tr>
<td>ExpInt</td>
<td>Exponential integral function</td>
<td>108</td>
<td>3</td>
<td>100</td>
<td>2.2</td>
</tr>
<tr>
<td>FDCT</td>
<td>Fast Discrete Cosine Transform</td>
<td>223</td>
<td>2</td>
<td>100</td>
<td>5.8</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>Iterative Fibonacci calculation</td>
<td>37</td>
<td>1</td>
<td>100</td>
<td>2.0</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>Insertion sort program</td>
<td>60</td>
<td>2</td>
<td>50</td>
<td>2.2</td>
</tr>
<tr>
<td>JanneComplex</td>
<td>Complex nested loops</td>
<td>72</td>
<td>2</td>
<td>0</td>
<td>3.3</td>
</tr>
<tr>
<td>MatrixCount</td>
<td>Count numbers in a matrix</td>
<td>85</td>
<td>4</td>
<td>100</td>
<td>2.9</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>Matrix multiplication</td>
<td>104</td>
<td>5</td>
<td>100</td>
<td>2.7</td>
</tr>
<tr>
<td>NestedSearch</td>
<td>Search in a multi-dimensional array</td>
<td>487</td>
<td>4</td>
<td>100</td>
<td>38.9</td>
</tr>
<tr>
<td>PetriNet</td>
<td>Petri Net simulation</td>
<td>4008</td>
<td>1</td>
<td>100</td>
<td>30.4</td>
</tr>
<tr>
<td>QuickSort</td>
<td>Quicksort program</td>
<td>166</td>
<td>6</td>
<td>0</td>
<td>4.7</td>
</tr>
<tr>
<td>Select</td>
<td>Select smallest element from array</td>
<td>136</td>
<td>4</td>
<td>0</td>
<td>3.8</td>
</tr>
<tr>
<td>SLE</td>
<td>Simultaneous Linear Equations</td>
<td>128</td>
<td>11</td>
<td>64</td>
<td>2.7</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>61</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Benchmark examples

the analysis. The analysis time was measured with the Java Hotspot VM (version 1.6.0), with a maximum heap size of 1.25 GB, running on an Intel Core 2 Duo CPU (model T7700) at 2.4 GHz. On average, 61% of the loops could be bounded. The results for the Mälardalen benchmarks are similar to the results presented in [14]. For the benchmarks derived from industrial applications, the percentage of detected loop bounds is close to the average. The analysis finishes within a few seconds for most benchmarks. However, Kfl and NestedSearch require a disproportional amount of time, when comparing the analysis time with the program size.

The DFA tool has been integrated into two WCET analysis tools [16, 9]. The WCET analysis tools use both the results of the DFA and manual annotations. This allows programmers to annotate loop bounds where the DFA is unable to compute one. It also makes it possible to detect inconsistencies between DFA results and manual annotations.

Although being a rather simple analysis, we found the loop bound analysis to be very useful in the context of automating WCET analysis. Several annotation errors in the test programs could be detected. In the Kfl benchmark, the annotated bounds for a few loops were too high. Such errors result in less tight WCET estimates, but are otherwise harmless. More serious annotation bugs were found in the Lift and SVM benchmarks where some of the annotated loop bounds were too low. Obviously, the WCET estimates computed from these annotations were not safe, and could have cause a system failure in the worst case.
4.1 Programming Idioms

While some of the undetected loop bounds for the benchmarks in Table 2 were simply algorithmically too complex for our analysis (e.g., a binary search on an array cannot be bounded), other loop bounds appeared to be undetectable due to certain programming idioms.

One operation that is difficult to handle for an interval based value analysis is equality/inequality. The reason for this is that this operation does not have a useful definition on intervals. A condition \( x \neq 0 \) is true for \( x \in (\left[ -\infty, -1 \right] \cup \left[ 1, \infty \right]) \). Merging these two sub-intervals into a single interval again yields \( [-\infty, \infty] \), i.e., no information has been gained from the operation. In contrast, less-than/greater-than operations in loop exit conditions help the analysis to find a bound, because they narrow down the possible values for the loop variable.

A related issue is the fact that our loop bound analysis cannot handle boolean flags for loop exit conditions. When using a boolean flag, the analysis cannot detect a sensible increment, and therefore assumes that the loop is unbounded.

A second problematic issue for a loop bound analysis is when values are passed via arrays. To limit the complexity of the analysis, individual array elements are usually merged into a single abstract value. In the case of the UdpIp benchmark, a message buffer array contained both the length of a packet and the packet data. As the packet data can contain arbitrary data, the packet length appears to the analysis to have an arbitrary value as well. Loops that use the packet length as bound can therefore not be bounded automatically.

Some applications use loops that depend on the environment, e.g., a timer reaching a certain value. Although it may be guaranteed that the program eventually exits the loop, the number of loop iterations until it does so is not specified inside the program logic. For such cases, it is inevitable that there exists some mechanism for a developer to specify a loop bound manually.

5 Related Work

As already discussed in Section 3.5.2, our approach to receiver types analysis is similar to the k-CFA [17, 18] analysis. Techniques like the ones described in [3] or [19] are more efficient than our approach, but are less precise. The differences stem from the fact that these analyses use larger scopes to compute information—in the extreme case, a global set of receiver types is computed. Differences also come from the fact that some analyses start from a conservative estimate of the call graph, while our approach builds up the call graph from scratch.

Our approach to loop bound analysis is similar to one of the flavors of abstract interpretation presented by Gustafsson in [5], namely the “approximate interpretation with widening/narrowing”. However, our approach differs in the way values are widened/narrowed, as it is tailored to the needs of our analysis. The prototype tool developed by Gustafsson does not use the widening/narrowing technique.

In a different (but somewhat similar) approach, Gustafsson et al. use abstract execution to detect loop bounds and infeasible paths [6]. With this approach, loop iterations are analyzed individually. On the one hand, this potentially yields very precise results. On the other hand, some of the potential precision must be sacrificed to keep the cost of the analysis feasible.

Prantl [14] extends a simple interval analysis with constraint solving to find loop bounds. The strength of that approach is that it naturally handles nested loops, a feature that lacks from usual loop bound analyses. The benchmarks in the paper indicate that a loop bound analysis can detect
more loops than the constraint solving approach. Unfortunately, no numbers on the tightness of the identified bounds are included.

In [4], a pattern-based and a data-flow-based approach to find loop bounds are compared. The data-flow-based approach only handles constant additions for loop variables. In contrast, our approach handles additions of bounded values and can therefore be considered slightly more powerful. The comparison to the pattern-based approach shows that the data-flow based approach is superior in most cases. Only in a few special cases, the former approach finds a bound where the latter approach cannot.

The approach of Healy et al. [8] computes information that is fairly similar to the information computed by our loop bound analysis. However, the information is only computed for induction variables. Unlike our approach, it also uses summation to handle nested loops and is therefore superior in this regard.

6 Conclusion and Outlook

In this paper, we presented a framework for analyzing Java bytecode. Building on this framework, two analyses were implemented: a receiver type analysis and a loop bound analysis. The latter analysis supports the WCET analysis of programs by removing the need for many manual annotations. The evaluation of a number of synthetic and application benchmarks showed that about two thirds of the loop bounds can be detected automatically. Moreover, programming idioms that pose problems for automated detection of loop bounds were identified.

The current receiver type analysis is precise, but inefficient. Consequently, future work to consider is the implementation of a more efficient analysis, such that large applications can be handled with reasonable effort. Some infeasible paths are detected by the loop bound analysis, but not exposed to the tool chain. On the one hand, infeasible paths can be used to optimize a program, on the other hand, WCET analysis is more precise if certain paths can be excluded. A future revision of the analysis implementation should therefore provide means to access this information.

Acknowledgement

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

The author thanks Benedikt Huber, who tested the framework and provided useful comments on early drafts of this paper.

References


