

Using Global Data Flow Analysis on Bytecode to Aid Worst Case Execution Time Analysis for Realtime Java Programs

James J. Hunt,
Isabel Tonin, and
Fridtjof B. Siebert
aicas GmbH
Haid-und-Neu-Straße 18
D-76139 Karlsruhe, Germany
jjh|tonin|siebert@aicas.com

ABSTRACT

Though realtime Java offers significant advantages over other programming languages for safe programming, the analysis of worst case execution of realtime Java programs is considerably more difficult. The extra complexity can be addressed using a minimal set of parameterized annotations and data flow analysis to provide a standard worst case execution time analysis tool with the additional information necessary to determine the worst case execution time analysis of realtime Java programs. This methodology has the advantage over existing methods in that it is equally applicable to general purpose library code as to application specific implementation code.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification

General Terms

Algorithms, Performance, Reliability, Theory, Verification

Keywords

Realtime Java, worst case execution time, data flow analysis, formal analysis, verification

1. INTRODUCTION

Many embedded applications rely on realtime behavior to do their job correctly. For example, an automatic stabilization system needs to react to all changes in position and attitude of the object being controlled. The reaction must occur reliably within a given bound. Ensuring that the control software reacts consistently within the required bounds requires detailed timing analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The 6th Workshop on Java Technologies for Realtime and Embedded Systems—JTRES '2008 Palo Alto, CA, USA
Copyright 2008 ACM 978-1-60558-337-2/08/9 ...\$5.00.

The first such control applications used single threading, simple processors, and procedural programming languages. This kept the analysis simple but also limited the complexity of the control application. As the requirements of control systems become more complex, this analysis becomes increasingly difficult.

Increased complexity results from three main technology trends: more complex processors, multithreading, and a more dynamic programming model. The problems of analyzing increasing complex processors is language independent and must be handled at the machine code level. Similarly, the scheduling analysis necessary for ensuring that a given task receives enough process time to complete within its deadline is also largely language independent. This discussion will focus on language issues that need to be considered when preparing the constraint information needed to drive the low level worst case execution time (WCET) analysis, in particular, analyzing realtime Java programs.

Java technology along with the Real-Time Specification for Java (RTSJ) offers significant advantages for programming complex systems at the cost of adding features that complicate timing analysis. Unless one is using a CPU that executes Java byte code directly, one needs to analyze the interpreter or the machine code generated from the byte code to machine code compiler. The most obvious language dependent feature needing special treatment is dynamic dispatch, but determining loop bounds is also necessary. Finally, the garbage collector needs to be taken into account. Given a static byte code compiler and an appropriate realtime garbage collector, the authors present a methodology for using a standard WCET analysis tool to calculate the maximum time needed to execute a time critical method in a realtime Java application.

Standard WCET analysis tools for a given processor are difficult to write, both due to the complexity of modern CPUs and the dearth of accurate information. Not only that, they lack high level information about loop bound, dynamic dispatch, memory allocation, and garbage collection. Most of these tools support some annotations, but these are usually quite concrete, e.g., the loop label `max_iterations=7` or a list of dispatch targets. They are also often expressed in separate annotation files with address offset references to machine code as well.

By combining such a tool, with data flow analysis, a bit of call context independent annotations, and some Java virtual machine (JVM) specific information, one can analyze quite

complex program segments, e.g. an RTSJ event handler. Data flow analysis is needed for two things: ascertaining the set of methods that may be called, i.e. the dispatch set of the call, and resolving the context independent bounds annotation to concrete bounds at each call point. Allocation and work based garbage collection are handled with JVM specific information in the same way that OS calls are handled. A high level analysis tool can take advantage of annotation files by generating concrete bound from data flow analysis and context independent annotations. Putting the parts together provide a complete view of the timing requirements of critical code without having to resort to unverifiable user input.

2. STATE OF THE ART

WCET analysis has been a part of realtime system design for quite a long time, but even here literature is relatively young. Before the advent of caching and pipelines in embedded processors, it was common practice to calculate WCET by hand using a table of instruction execution times given in clock cycles. Static analysis papers started to appear at the end of the 80's, e.g. the work of Messrs. Puschner and Koza[15].

Most of the activity in the field is centered in Europe, where there are still a handful of groups active including the TU Vienna in Austria, Saarbrücken in Germany, INRIA/IRISA in France, the University of York in the UK, and the ARTES network of the Uppsala University, the Mälardalen University, Chalmers University of Technology, and Lund Institute of Technology, all in Sweden. Unfortunately, being research projects, very few tools are really usable beyond the laboratories in which they were designed. Information about them can only be inferred from the papers written about them. A contributing factor to the dearth of information is the hope of marketing the tools and the consequent desire to limit free access.

There are many interesting aspects of WCET analysis, even Java processors designed to simplify WCET analysis, but little germane to handling loop and recursion bounds or dynamic dispatch. Much of the work has concentrated on machine dependent aspects of WCET analysis, such as pipeline, cache, branch prediction, and instruction dispatch analysis. Much less has been done with machine independent aspect of analysis, such as loop bounds and dispatch set analysis.

2.1 CALC_WCET_167

CALC_WCET_167 [11] was developed in the Real-Time Systems Research Group at the Vienna University of Technology. This tool is designed to provide WCET analysis on the Siemens C167CR architecture (hence the name). It is tightly linked with a special version of GCC (the GNU C Compiler) for the C167 which uses WCETC [12], a derivative of the C programming language, as its input language. WCETC includes support for specifying the loop bound on the loop headers (do..while, while, and for loops) and the maximum execution count of certain paths in the flow graph. It forbids the use of function pointers, recursive function calls, goto, setjmp(), longjmp(), signal() and exit().

2.2 Cinderella 3.0

One of the first tools to be able to model advanced processor architectures, Cinderella[14] has some interesting fea-

tures, as it was designed to be easily retargeted to other architectures. The author provides architecture models for the Intel i960 and the Motorola M68000. Cinderella performs its analysis on the executable files in COFF format in both architectures, but it requires the user to specify via the provided GUI the minimum and maximum loop iteration bounds.

2.3 Heptane

Heptane¹[5, 6] was developed by IRISA as part of the Hades project². Heptane is a fairly complete tool that models a very recent processor: the Intel Pentium. Heptane is built on top of Hexane, a preprocessor that reads the C source file and outputs the syntactic tree of the program under analysis, and Salto, which provides the hardware dependent information. Maximum loop iteration is provided by the user in the form of source code annotations. Some restrictions on the source code are that the use of function pointers is forbidden and features that may cause unstructured control flow graphs (i.e. control flow graphs that are not compatible with the syntax tree) are not allowed including the use of the goto construct and the inclusion of assembly code that contains branching instructions.

2.4 aiT

aiT[7] is a fairly complete tool produced by AbsInt, in close cooperation with the Universität des Saarlandes in Saarbrücken. It takes as input the executable file in ELF or COFF formats. Dynamic data structures, and setjmp(), longjmp() calls can not be analyzed. aiT tries to determine the number of loop iterations by loop bound analysis, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided as user annotations. Loop bound analysis relies on a combination of value analysis and pattern matching, which looks for typical loop patterns. aiT assumes that the generated code is well behaved. In general, these loop patterns depend on the code generator and/or compiler used to generate the code that is being analyzed. There are special aiT versions adapted to various generators and compilers.

2.5 Bound-T

Bound-T[1] is a tool developed by Space Systems Finland mostly through ESA (European Space Agency) funded projects. Similarly to aiT, Bound-T analyzes compiled and linked executables and provides some automatic analysis of loop bounds. During the arithmetic analysis of a subprogram, Bound-T finds the potential loop counter variables for each loop and tries to bound the initial value, the step value (increment or decrement), and the limit (loop terminating) value of each potential loop counter. If it succeeds, it bounds the number of repetitions of the loop. If there are several loop counters for the same loop, Bound-T uses the one that gives the least number of repetitions. When this process does not work, the user must supply annotations.

2.6 Gromit

The Microelectronic System Design group in the Forschungszentrum Informatik an der Universität Karlsruhe in Germany has built a WCET tool that focus on two PowerPC processors: the PPC403, the MPC750[8, 9]. This

¹Hades Embedded Processor Timing ANalyzer

²Hard real-time DEpendable Systems

tool, called Gromit, was written in Java with some auxiliary modules in C++. It relies on bound annotations given in auxiliary files. In the HIJA project, this was the base for investigating using information from data flow analysis and runtime monitoring.

2.7 JOP

Messrs Schöberl and Pedersen[16] have presented a WCET analysis tool for Java processors. The central idea is to adopt the processor to realtime Java and WCET analysis rather than the analysis to Java and the processor. The approach simplifies the low level analysis, but does not advance high level analysis.

2.8 Modes

The WCETAn Tool from the University of York [3, 4] aims to provide portable WCET analysis for the Java programming languages by associating timing information with the Java bytecodes in the code under analysis. This tool also uses annotations to bound loops but introduces the concept of modes to give more flexibility on bounds. A given method may have a predefined number of modes, each with its own loop fixed bound. Each call point is associated with a mode. Thus the bound used is determined by the call point, though only a finite set of modes can be defined.

2.9 Analysis

Tools	annotation	automatic
CW_167	constant	no
Cinderella 3.0	constant	no
Heptane	constant	no
aiT	constant	simple loops only
Bound-T	constant	simple loops only
Gromit	constant	not really
JOP	constant	no
WCETAn	modes	no

Table 1: Loop bounds handling in current Tools

As depicted in Table 1, all current tools depend on user provided loop bounds, whether flexible as with modes or constant in other tools. Even the tools which perform analysis to find constant loop bounds need annotations in most cases. Current tools pose two problems: there is no mechanism for determining the correctness of the bound given and the bounds themselves are inflexible. Even execution modes suffer from both drawbacks, since even there only a finite set of modes can be provided. Unless the bound of the loop is also a constant, these kinds of annotation with concrete bounds drastically limits code reuse. Often bounds are determined by the size of some collection, so when the given collection is not defined directly in the method, the bounds can not be know locally.

3. METHODOLOGY

Object oriented programming in general and realtime Java programs in specific tend to use more dynamic constructs than the procedural programming typically used in time critical programs. Loop and recursion bounds are not usually determinable with local information. The target of dynamic methods calls are also data dependent. In order to employ standard analysis tools for WCET calculation, these

additional complexities must be addressed. Fortunately, the Java byte code offers a convenient representation for the data flow analysis necessary for tracking the non local information needed for capturing data dependencies. By combining parametrized loop and recursion bounds annotations with data flow analysis, one can sufficiently characterize the dynamic behavior of object oriented programs to be able to use current analysis tools, aiT in this case, to calculate the WCET of critical code segments written for a realtime Java program.

3.1 Data Flow Analysis

Data flow analysis is a technique for collecting the possible set of values calculated at various points in a program. A control flow graph is constructed for determining those parts of a program to which each given value might propagate. For an object oriented program, these value set contain the object references through which methods calls are dispatched. They also contain the bounds information needed for loops and recursion.

The data flow analysis tool, Veriflux, used here in collecting dynamic context information for high level WCET analysis is an extension of the context and flow sensitive program analysis described previously[18]. Analysis is made on a static single assignment intermediate representation derived from Java byte code. Complex instructions are broken into parts and the JVM stack and registers have been removed. Pointer analysis has been augmented with simple arithmetic analysis to maintain integer ranges as well as pointer sets. This additional information is often necessary for determining loop bounds.

During execution, Veriflux, determines two sets of values with contexts: the set of invocations and the set of reference values for each referenced field and referenced array element. The analysis runs iteratively, tracing the call graph starting with main. The data flow for all invocations is analyzed in each iteration. The algorithm terminates when these sets remained constant during one iteration. The analysis distinguishes object initialization and object use to provide better accuracy. Specific properties, such as singleton instances and embedded instances, are also detected to further improve analysis accuracy. For each method call, context information enables the tool to calculate their dispatch sets, i.e., the set of actual methods to which the call can be dispatched.

3.2 Loop Bounds

The bounds of most loops are data dependent. This is particularly true of library code. In order to handle library code fully, parametrized bounds are needed. For this a standard notation is desirable.

The Java Modeling Language (JML)[13] defines a clause, **decreases** <expression>, which can be used to specify the bound of a loop. Using JML annotation has the advantage that the bounds can be formally verified and is consistent with annotations for functional verification as the authors have shown previously[10]. The value of the expression must decrease by at least one for each iteration of the loop and the loop terminates when the expression reaches zero. The expression itself is not the bound of the loop, rather the loop is bounded by the maximum value that the expression can take. By annotating all critical loops with this clause and evaluating each **decreases** expression within the data flow

analysis just before its associated loop, a maximum value can be obtained for the associated loop in the given call context.

Each `decreases` expression is passed to the analysis by code injection. In other words, JML annotated source code is automatically translated into a form that enables the data flow analysis to record the value of the `decreases` clause when the loop is entered. The value of all arguments are saved with each call point, which means that the single `int` argument to `DFAHelper.captureBounds` is recorded at each call point in the call's context.

This transformation is invisible to the user and is only done at the source level because one can leverage the standard JML parser to do most of the work. It could just as well be done at the bytecode level, had the decreasing clause been captured as an annotation; however, a more extensive infrastructure would be needed. The `DFAHelper.captureBounds` method itself is more of a place holder than an annotation. `DFAHelper.captureBounds` is a public final static method that does nothing.

There are three general cases to consider, while loops, for loops, and for each loops. In each case, a block of semantically neutral code is added just before the loop. The expression in the JML clause is taken as the argument to `DFAHelper.captureBounds`. Enough context is provided to ensure that all variables used in the expression have their proper values.

For both forms of while loops, all variables that can be used in the `decreases` expression must have been defined before the loop is entered. Since the expression may not have side effect, it can be inserted into the code. For instance, the method fragment

```
\\@ decreases elements.length - i;
while (i < elements.length)
{
    sum += elements[i++];
}
```

would be converted by the tool into

```
{
    DFAHelper.captureBounds(elements.length - i);
}
while (i < elements.length)
{
    sum += elements[i++];
}
```

For for loops, variables that can be used in the `decreases` expression can be defined in the loop header. This means that any variables defined in the loop header must be defined in the injected block as well. Thus, the tool would automatically convert the method fragment

```
\\@ decreases elements.length - i;
for (int i = 0; i < elements.length; i++)
{
    sum += elements[i];
}
```

into

```
{
    int i = 0;
    DFAHelper.captureBounds(elements.length - i);
```

```
}
for (int i = 0; i < elements.length; i++)
{
    sum += elements[i];
}
```

where `int i = 0` is taken from the loop header creating the same value in the injected block as at the start of the loop.

Writing a `decreases` clause for a for each loop would be a bit more difficult, since a ghost variable would be necessary. For example,

```
\\@ ghost int i = elements.length;
\\@ decreases i;
for (elements: entry)
{
    sum += elements[entry];
    \\@ set i--;
}
```

would be converted by the tool into

```
{
    int i = elements.length;
    DFAHelper.captureBounds(i);
}
for (elements: entry)
{
    sum += elements[i];
}
```

where the `set` statement need not be reflected in the code. However, this is not necessary, since a for each always iterates exactly as many times as it has elements. Thus,

```
for (elements: entry)
{
    sum += elements[entry];
}
```

would be automatically converted into

```
{
    DFAHelper.captureBounds(elements.length);
}
for (int i = 0; i < elements.length; i++)
{
    sum += elements[i];
}
```

without the need for annotations at all.

One might object that the modified code is not exactly the same as the original, thus making certification more complex; but this objection is without merit. The injected code has no semantic effect on the results. A good optimizer will remove the extra code, so the same source can be used both for analysis and production without any loss of efficiency. Also, the fact that the correctness of any given `decreases` clause may be determined formally^[10] is an added advantage for certification.

3.3 Recursion Bounds

Recursion can be annotated and analyzed similarly. The corresponding clause is `measured_by <expression>`³. Analogous to `decreases`, the value of the expression must diminish by at least one at each deeper recursive call and the recursion ends when the expression reaches zero. Thus

³Since `measured_by` has semantics analogous to `decreases`, `measured_by` may be renamed in JML to `decreases`

```

public int sum(int[] elements)
{
    int sum = 0;
    int index = elements.length;
    return sum(elements, index, sum);
}

/*@ measured_by index;
private int sum(int[] elements, int index, int sum)
{
    if (index == 0)
        return sum;
    else
    {
        sum += elements[--index];
        return sum(elements, index; sum);
    }
}

```

becomes

```

public int sum(int[] elements)
{
    int sum = 0;
    int index = elements.length;
    {
        DFAHelper.captureBounds(index);
    }
    return sum(elements, index, sum);
}

/*@ measured_by index;
private int sum(int[] elements, int index, int sum)
{
    if (index == 0)
        return sum;
    else
    {
        sum += elements[--index];
        return sum(elements, index; sum);
    }
}

```

where the code for `measured_by` is injected before the first call to the recursive function.

This is more complex than the loop bounds case, since the code injection is not local. An alternative would be to put the injected code into the recursive function, but this would move the complexity. Often, there is some guard function, as for the tail recursive case above, that limits the extent of the non locality.

One would expect that a deductive formal verification tool such as KEY [2] could be used to verify the correctness of these annotations as well.

3.4 Dispatch Sets

Unlike procedural code, the call graph of an object oriented can not be generated without considering data coupling. The destination of each call point is determined dynamically. For WCET analysis, all possible destinations of a method call must be known.

Annotations do not really help in determining these dispatch sets. Source code annotations are too coarse, particularly for library code. Finer annotations would hard to

relate to specific call points. Demonstrating the correctness of such annotations would also be challenging.

These dispatch sets can be determined using a global data flow analysis tool such as Veriflux. Accurate results require as much context for each call point as possible, but not so much that the data size explodes. In Veriflux, a call point is not simply the `Invoke` in a particular method, but also the allocation context of the objects in its argument list including `this` and the current thread.

This means that, given an entry method to analyze for WCET analysis, and the context set generated from a full program data flow analysis, the call sets for each call point can be determined to a higher accuracy than source code annotations would enable. This is because data flow analysis can keep track of different callpoint containing a given method with totally different dispatch sets. Source code annotations would have to contain the union of all possible sets for the given method.

Of course for accurate WCET calculations, the dispatch effort itself must be characterized for the JVM. In JAMAICAVM, this is just a fixed index into the method table of the object on which the method is called. The dispatch set is translated into a target list annotation for aiT designed for indirect procedure calls, since method dispatching in JAMAICAVM is exactly this.

3.5 Allocation

Allocation in the JAMAICAVM are handled by standard routines of the Jamaica runtime environment. As such, the aiT annotations needed for their analysis have been done by hand and their WCET precalculated: both for String allocation and another for all other objects. These routines would have to be redone for use with another tool.

3.6 Realtime Garbage Collection

How realtime garbage collection must be handled for WCET analysis is intimately dependent on how the garbage collector works. There are a few variants of mark and sweep collectors used by realtime Java implementations:

- time based collectors, such as Metronome and PERC, which schedule GC work periodically;
- slack based collectors, such as McKinac, which scheduled GC work in the slack of the RT tasks; and
- work based collectors, such as JAMAICAVM's collector, which combine GC work with allocation work.

The first two variants must not only be analyzed for the WCET of their GC work increments, but must also be considered in scheduling. Work based collectors can be designed to be completely run in the allocating thread, so that only their impact on WCET need be considered. Worked base collectors with fixed allocation blocks, such as JAMAICAVM's, also ease WCET analysis since there are only very simple mark and sweep steps and no compaction. The discussion here is limited to in thread collectors.

The JAMAICAVM garbage collector is only active when new memory is needed, i.e., each block of allocation is payed for by doing a few small increments of garbage collection work. This means that the GC work is always done when some object is allocated and therefore can be rolled into the WCET. The collector is incremental to the extreme: single incremental steps of the garbage collector correspond

to scanning just 32 bytes of memory and have a WCET on the order of one μsec .

For every block that is allocated, the number of garbage collection increments that are performed is $P(F) = M/F$, where M is the size of the heap (in number of blocks) and F is the number of blocks that are currently free. The garbage collector guarantees finishing one cycle after the number of GC increments (the sum of all $P(F)$) exceeds the number of blocks allocated at the end of the cycle. Since a mark and sweep collector requires two scans of the allocated memory, one during the mark phase and one during the sweep phase, each GC increment corresponds to two mark or sweep steps, depending on the phase the garbage collector is in.

Performing $P(F)$ increments of GC work for each block of memory allocated guarantees sufficient garbage collection progress as long as the amount of reachable memory stays below the system's total memory. Furthermore, a worst case upper bound P_{max} for P can be determined if the amount of reachable memory R stays below a fixed upper bound K (such that $R \leq K$ always holds). This upper bound also gives an upper bound for the time required to allocate one block of memory.

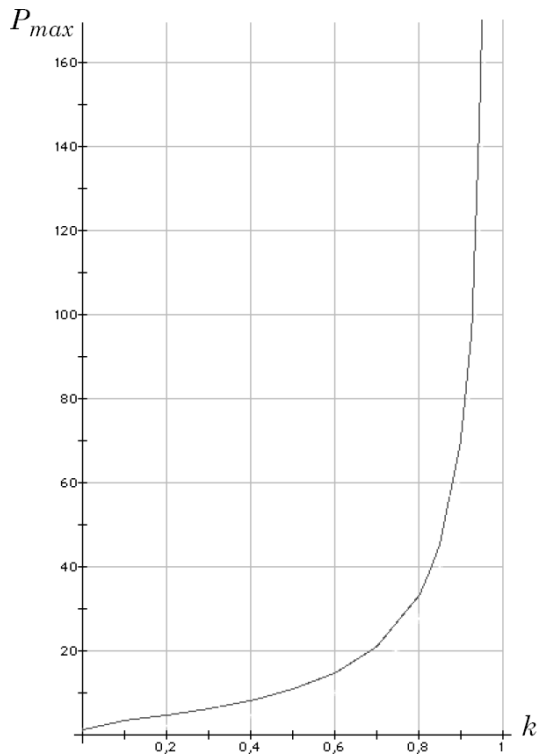


Figure 1: Worst-case number of GC increments (mark and sweep work units) required for the allocation of one block as function of k of the maximum percentage of reachable memory $k = K/M$.

An upper bound on the maximum amount of reachable memory K should be known by the developer; though, determining the upper bound on the amount of reachable memory is a difficult task for non trivial applications. A derivation of the dependency between the upper bound on reachable memory K and the upper bound for the GC work P_{max} is beyond the scope of this article. Detail information can

be found in "Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages" [17]. Though the detail may vary, the basic exponential relationship between available memory and the amount garbage collection work needed to be done, as depicted in Figure 1, applies to all garbage collectors.

The mark increment and the sweep increment can be analyzed separately for each processing platform. Then, Application analysis must provide the number of blocks for each allocation and the number of units of GC work to perform per block. These values combined with the WCET of one unit of GC work provides the WCET GC overhead for each allocation.

Along with the allocation overhead, the system also considers write barriers and synchronization points. There are no read barriers, which would occur much more often. The write barriers are completely handled directly by aiT, since they are coded inline. The synchronization points are handled similarly, but the actual switch itself is not considered, since its cost is part of the task switch cost for scheduling analysis.

3.7 Building Data Contexts

In general, it is not realistic or necessary to analyze an entire application for WCET, but rather analysis is focused on critical tasks with a defined deadline by which it must complete. Meeting this deadline is dependent both on the actual execution time needed to perform the task and the amount of CPU time allocated to the task within the deadline. WCET analysis focuses on the former; whereas, the latter is the domain of scheduling analysis.

The system architect must have some notion of how much CPU time is available to the task within its deadline. This time is often confused with WCET in the literature, but the authors consider this to be the time budget for the task. The acceptance criteria is that the WCET, an analysis result, is within the time budget for the task.

When analyzing an application with the technique described herein, one needs WCET results for an number of critical tasks. In general, the contexts needed for analysis must take the entire application in consideration. For this reason, the data flow analysis is first performed on the entire application, then the critical routines are analyzed using the value sets generated by the data flow analysis. The analysis is predicated on having an entry method, such as the `handleEvent()` method of an `AsyncEventHandler` or a method called by from a `waitForNextPeriod()` loop in a `RealtimeThread` with `PeriodicParameters`. For test purposes during development, one can use an abbreviated program that just sets up the proper context for a particular method, enabling one to start analysis before the entire program is available. Still, final analysis needs to be done with the full application.

4. EXAMPLE

An an example, one can use a method from a safety critical avionics application: the `mergeProcedures` method as depicted in Figure 2. This example has been presented in a previous paper[10]. The top of the figure shows the general view of the method: it receives two procedures, i.e. parts of a path an airplane ought to follow, and merges them producing a result procedure. This procedure containing some legs from the first and some legs from the second procedure,

where legs are partial paths. The way the procedures are merged is shown at the bottom of the figure. *legs1* and *legs2* represent the legs from the procedures 1 and 2 respectively (in this implementation *legs1* and *legs2* are arrays). The resultant procedure is built with legs from *legs1* (from the beginning until the point *i1*) and from *legs2* (from the point *i2* until the end). The calculation of the points *i1* and *i2* determines which legs will be part of the result procedure. These points are calculated as following.

- *i1* represents the last leg from *legs1* to be part of the result procedure. It is found by searching for an intersection point between *legs1* and *legs2*. Each leg from *legs1*, starting with the first, is tested to see whether it exists in *legs2* or not. If one exists, this legs becomes *i1*. If no intersection was found, the last leg of *legs1* becomes *i1*.
- *i2* represents the first leg from *legs2* to be part of the result procedure. Its calculation is performed within the calculation of point *i1*. When an intersection is found, the first leg of *legs2* after the intersection leg becomes *i2*. If no intersection is found, the first leg from *legs2* becomes *i2*.
- *i3* represents a limit for the search for intersection performed in *legs2*. It is the first occurrence of a leg of the type *runway*.
- The search for intersection in *legs2* is performed backwards (from the *i3* point to the beginning) in order to find the last intersection point before the runway.

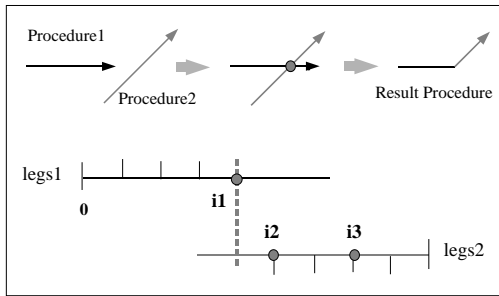


Figure 2: The mergeProcedures method analysis.

The method contains five loops (two of them nested) subdivided in three submethods.

- **indexFirstRunway** calculates the point *i3*. The *indexFirstRunway* method annotated for WCET analysis is shown in Figure 3. It contains one loop over *legs2*, whose bound is the size of this array, as expressed in the decrease clause expression (*legs2.length - j*) (since the initial value of the variable *j* is zero).
- **findIntersection** calculates the points *i1* and *i2*. The *findIntersection* method annotated for WCET analysis is shown in Figure 4. It contains a nesting of two loops: for each element *x1* of *legs1* an element *x2* is search backwards in *legs2* from the point *i3*, such that ($x1 = x2$). The bound for the first loop is the length of the first array, as shown in the decrease clause expression

```
private static int indexFirstRunway(Leg[] legs2)
{
    int j = 0; Leg leg;
    //@ decreases (legs2.length - j);
    while (j < legs2.length)
    {
        leg = legs2[j];
        if (leg instanceof KFixLeg)
        {
            if (((FixLeg)leg).getFix() instanceof Runway);
            return j;
        }
        j++;
    }
    return legs2.length - 1;
}
```

Figure 3: indexFirstRunway annotated for WCET analysis

(*legs1.length - j*). The bound of the second loop is the number of elements in the second array, from zero up to the point *i3* (in the code *k*), as in the decrees clause expression (*k + 1*).

- **finishMerging** generates the result procedure (array). The *finishMerging* method annotated for WCET analysis is shown in Figure 5. It contains two loops. In the first loop legs from *legs1*, from the position zero until *i1*, are copied to the result array. The bound of this loop is thus represented by the expression (*i1 - j + 1*) where *i1* is a variable that holds the position *i1*. Finally, in the second loop, legs from *legs2* from *i2* until the end are copied to the result array. The bound for this loop, in the code, is given by the expression (*merge.length - k*) where *merge.length* is the length of the resulting array and *k*'s initial value is the position *i1* plus one.

Using the value sets generated by data flow analysis, which contain all the arrays that can be processed by this code, the concrete bound can be derived in the context of the full application.

By augmenting the annotations a bit, one can prove the correctness of the given loop bounds. The additional annotations needed for the method *findIntersection* are shown in Figure 6. Verification of the bound requires a minimal contract for the method, so that termination of the method can be proven.

In order to functionally verify these methods, full contracts have to be written expressing the methods' behaviors in terms of preconditions and postconditions for each method. In addition, each loop would also have to be fully specified. Figure 7 gives an example for the *findIntersection* method.

Though the annotation required for full functional verification are quite involved, one can start with quite simple annotations for WCET analysis and refine them later if desired. By using JML, the refinements need not introduce any redundant information. If Java annotations where powerful enough, JML could be recast to take advantage of the ability to carry annotations along in byte code, but this would require some sort of statement level annotation extension.

```

private void
  findIntersection(Leg[] legs1, Leg[] legs2, int i3)
{
  i1= legs1.length - 1;
  i2 = 0;
  int j = 0;
  int k = i3;
  Leg leg;
  //@ decreases (legs1.length - j);
  while (j < legs1.length)
  {
    leg = legs1[j];
    if (leg instanceof KFixLeg)
    {
      k = i3;
      //@ decreases (k + 1);
      while (k >= 0)
      {
        if (leg == legs2[k])
        {
          i1 = j;
          i2 = k + 1;
          return;
        }
        k--;
      }
      j++;
    }
  }
}

```

Figure 4: findIntersection annotated for WCET analysis.

```

private
  Leg[] finishMerging(Leg[] legs1, Leg[] legs2)
{
  int diff = i2 - i1 - 1;
  Leg[] merge =
    new Leg[(i1 + 1 + (legs2.length - i2))];
  int j = 0; int k = i1 + 1;
  //@ decreases (i1 - j + 1);
  while (j <= i1)
  {
    merge[j] = legs1[j];
    j++;
  }
  //@ decreases (merge.length - k);
  while (k < merge.length)
  {
    merge[k] = legs2[k + diff];
    k++;
  }
  return merge;
}

```

Figure 5: finishMerging annotated for WCET analysis.

5. CONCLUSION

Leveraging data flow analysis to provide more complete and verifiable input to standard WCET analysis tools such as aiT, bring WCET analysis to realtime Java applications. This technique reduces the annotation burden for object oriented programs to context independent loop and recursion annotations while improving reusability, since the annotations needed are not application specific. With additional annotations, these bounds themselves can be formally ver-

```

/*@ private normal_behavior
  @ requires legs1!=null && legs1.length>0 &&
  @       legs2!=null && legs2.length>0 &&
  @       i3>0 && i3<legs2.length ;
  @ assignable i1, i2;
  @ ensures true;
  @*/
private void
  findIntersection(Leg[] legs1, Leg[] legs2, int i3)
{
  i1 = legs1.length - 1;
  i2 = 0;
  int j = 0;
  int k = i3;
  Leg leg;
  /*@ assignable leg, j, k, i1, i2;
  @ decreases (legs1.length - j);
  @*/
  while (j < legs1.length)
  {
    leg = legs1[j];
    if (leg instanceof KFixLeg)
    {
      k = i3;
      /*@ assignable k, i1, i2;
      @ decreases (k + 1);
      @*/
      while (k >= 0)
      {
        if (leg == legs2[k])
        {
          i1 = j;
          i2 = k + 1;
          return;
        }
        k--;
      }
      j++;
    }
  }
}

```

Figure 6: findIntersection annotated for loop bound verification.

ified. Notwithstanding, the required annotations are completely compatible with functional verification, providing the user with a uniform set of annotations for all three tasks.

6. FUTURE WORK

Thus far, enough work has been done to yield WCET numbers for some test code, but these numbers have not yet been experimentally verified on a board that is supported by aiT. The next step will be to test some applications and compare WCET results with measured response times on a PowerPC based board. Integration of Veriflux, aiT, and eclipse is also ongoing. Beyond that, it would be interesting to combine this technique with other Java bases systems such as JOP. Extending Java annotations, so that JML could be recast as a set of Java annotation classes, would also be interesting to better support library code, since WCET analysis could then be done without source code.

7. ACKNOWLEDGMENTS

Some of the funding for this work was provided through the SuReal project sponsored by the German Government (BMBF) for which the authors are grateful.

```

/*@ private normal_behavior
@ requires legs1 != null && legs1.length > 0 &&
@     legs2 != null && legs2.length > 0 &&
@     i3 > 0 && i3 < legs2.length ;
@ assignable i1, i2;
@ ensures i2 == 0 && i1 == legs1.length - 1 ||
@     (i2 > 0 && i2 <= i3 + 1 &&
@     i1 >= 0 && i1 < legs1.length &&
@     legs1[i1] != null &&
@     legs1[i1] == legs2[i2 - 1]);
@*/
private void
findIntersection(Leg[] legs1, Leg[] legs2, int i3)
{
    i1 = legs1.length - 1; i2 = 0;
    int j = 0; int k = i3; Leg leg;
    /*@ loop_invariant
    @ j <= legs1.length && j >= 0 &&
    @ i2 == 0 && i1 == legs1.length - 1;
    @ assignable leg, j, k, i1, i2;
    @ decreases (legs1.length - j);
    @*/
    while (j < legs1.length)
    {
        leg = legs1[j];
        if (leg instanceof KFixLeg)
        {
            k = i3;
            /*@ loop_invariant
            @ k <= i3 &&
            @ ((k >= -1 && i2 == 0 &&
            @ i1 == legs1.length - 1) ||
            @ (j < legs1.length && j >= 0 &&
            @ k > -1 && i2 == k+1 &&
            @ i1 == j && legs1[j] !=null &&
            @ legs1[j] == legs2[k]));
            @ assignable k, i1, i2;
            @ decreases (k + 1);
            @*/
            while (k >= 0)
            {
                if (leg == legs2[k])
                {
                    i1 = j;
                    i2 = k + 1;
                    return;
                }
                k--;
            }
            j++;
        }
    }
}

```

Figure 7: findIntersection annotated for functional verification.

8. REFERENCES

- [1] Bound-t tool homepage. URL: <http://www.bound-t.com/>.
- [2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
- [3] G. Bernat. Javelin webpage. URL: <http://www-users.cs.york.ac.uk/~bernat/javelin/index.html>, Mar. 2000.
- [4] G. Bernat, A. Burns, and A. Wellings. Portable worst case execution time analysis using java byte code. In *Proc. 12th EUROMICRO conference on Real-time Systems*, June 2000.
- [5] A. Colin. Heptane webpage. URL: <http://www.irisa.fr/solidor/work/heptane-demo/heptane.html>, Feb. 2001.
- [6] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherlands, June 2001.
- [7] C. Ferdinand and R. Heckmann. ait: worst case execution time prediction by static program analysis. In R. Jacquart, editor, *IFIP Congress Topical Sessions*, pages 377–384. Kluwer, 2004.
- [8] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on modern processor architectures. In *Proceedings of the Date 2000 Conference*, pages 552–559, Paris, France, Mar. 2000.
- [9] A. Hergenhan, A. Siebenborn, and W. Rosenstiel. Studies on different modeling aspects for tight calculations of worst case execution time. In *WIP-Proceedings of the 21th IEEE Real-Time Systems Symposium*, Orlando FL, USA, Nov. 2000.
- [10] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *JTRES 2006: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 162–169, New York, NY, USA, 2006. ACM Press.
- [11] R. Kirner. calc-wcet.167 webpage. URL: http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/, Oct. 2001.
- [12] R. Kirner. The programming language wcetc. Research Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
- [13] G. T. Levens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual. <http://www.jmlspec.org/>, 2004.
- [14] Y.-T. S. Li. Cinderella 3.0 home page. URL: <http://www.ee.princeton.edu/~yauli/cinderella-3.0/>, Oct. 1996.
- [15] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, May 1989.
- [16] M. Schoeberl and R. Pedersen. Wcet analysis for a java processor. In *JTRES 2006: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 202–211, New York, NY, USA, 2006. ACM.
- [17] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.
- [18] F. B. Siebert. Proving the absence of RTSJ related runtime errors through data flow analysis. In *JTRES 2006: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 152–161, New York, NY, USA, 2006. ACM.