WCET Analysis of Reusable Portable Code

Peter Puschner*
Institut für Technische Informatik
Technische Universität Wien
A1040 Wien, Austria
Email: peter@vmars.tuwien.ac.at

Guillem Bernat
Department of Computer Science
University of York
York, YO10 5DD, United Kingdom
Email: bernat@cs.york.ac.uk

Abstract

Traditional worst-case execution-time analysis (WCET analysis) computes upper bounds for the execution times of code. This analysis uses knowledge about the execution context of the code and about the target architecture. In contrast, the WCET analysis for reusable and portable code has to abstract from parameters that are unknown until the code is finally used. The analysis is done in two steps. The first step computes abstract WCET information to support the reuse and portability of the WCET information. The second step uses the abstract WCET information to compute concrete WCET bounds when the application context and the timing parameters of the target system are known. The paper describes each of the two analysis steps. It demonstrates how WCET information can be made portable and reusable.

1 Introduction

Portability and reusability have been goals of software developers for a long time. Only lately, with the uprise of Java [5], its portable Java Byte Code format [9], and the steadily growing number of class files available, portability and code reuse have gained increased importance. To this time, however, portability and reusability have only been associated with the *functionality* of code. There has hardly been any consideration about reusing or porting information about the *execution time* of code.

This paper focusses on an approach that makes knowledge about the timing of code reusable and portable. Of course, one cannot expect that code would need the same time to execute in different applications or on different hardware platforms. By portable and reusable timing information we mean that information about the timing of the code is pre-processed and stored with the code in an abstract format which can be easily evaluated once the concrete application- and target parameters are known.

The proposed approach is based on a two-step WCET analysis: The first step, performed by the software developer, generates the abstract execution-time representation. The second step takes place at the user's target environment or even on the target (e.g., a Java enabled device). It instantiates the abstract execution-time representation with the parameters of the application and the target environment and evaluates the resulting expression. The resources at the target site are typically very limited. Therefore, this second step is kept very simple.

WCET analysis needs to deal with two types of information: The possible sequences of actions (instructions/statements) that can occur during code execution and the duration of each action during each of the executions [12]. While both areas are equally important to WCET analysis, this paper will focus on the first aspect. It also shows which information about possible action sequences (paths) is suited for being used in a WCET format for reusable and portable code and which restrictions on path information have to be imposed. Note that this goal is different from other work, e.g., [11, 4] which aims at a comprehensive description of execution path information.

As for the timing of the single actions we assume that a sensible (i.e., not too pessimistic) WCET bound can be computed for each instruction/statement from a very local context, e.g., by analyzing instructions of the basic block (and possibly neighboring blocks) containing the instruction/statement. Also we assume that there exists a simple additive rule as included in the approaches proposed by Shaw [14] or Puschner and

*The presented work was performed while the first author was on a research stay at the University of York. This stay was supported by an EU Marie-Curie Fellowship, under contract no. HPMF-CT-1999-00184.

Koza [13] to compute the WCET bound of sequences of instructions/statements from the WCET information about single instructions. We do, however, not discuss the details of this addition and suitable data formats for storing instruction timing in this paper, but use normal arithmetic addition instead (details on more powerful timing schema can be found in [7, 1]). Due to these assumptions the approach supports portability between architectures with and without instruction pipelines, but excludes the modelling of all types of caching.

The contributions of the paper are:

- A two-step WCET analysis that generates an abstract intermediate format for execution-time analysis and allows for a simple WCET evaluation on the final target.
- An abstract format for the application-independent and machine-independent representation of WCET information.
- Restrictions on the path information to be provided as input to the WCET analysis. We will show that we have to restrict path information to keep the second step of WCET analysis simple.

The paper is structured as follows. Section 2 gives an introduction to WCET analysis. Section 3 presents the two steps of our approach to a WCET analysis for reusable and portable code. It describes the information generated and processed by these two steps. The example in Section 4 illustrates the operation of the WCET method and Section 5 evaluates the method for three code examples. Section 6 concludes the paper.

2 WCET Analysis

Informally, WCET analysis is defined as follows:

> WCET analysis computes upper bounds for the execution times of pieces of code for a given application, where the *execution time of a piece of code* is defined as the time it takes the processor to execute that piece of code.

Note the following points about this informal definition of WCET analysis that are worth mentioning:

- WCET analysis computes *upper bounds* for the WCET, not necessarily the exact WCET.
- The WCET bound computed for a piece of code is *application-dependent*. As the execution paths through the code may differ between different applications, the same piece of code may have different WCETs and thus WCET bounds.
- WCET analysis is *hardware-dependent*.
- Although the above definition does not mention the quality of the results of WCET analysis, WCET analysis is usually required to deliver bounds that are close to real WCET values.

A variety of different WCET-computation strategies have been proposed. Some approaches put the efforts on exact modelling of hardware features like instruction pipelines and caches, e.g., [6, 10, 8]. Others focus on utilizing detailed path information in order to obtain tight WCET bounds, e.g., [11, 3, 4]. Most approaches are complex and do not support a simple WCET computation from an intermediate execution-time representation.

We therefore take the timing-schema approach [14] as the starting point for our WCET analysis. The timing-schema approach uses execution-time calculation rules for different constructs (simple, if-then-else, loop, etc.) and construct sequences. It applies these rules recursively, following the syntactic structure of the code to compute WCET bounds, see Table 1.

Table 1. Simple Timing Schema

S	$t_{max}(S)$
simple	WCET of S
S1; S2;	t_{max}(S1) + t_{max}(S2)
if (E) S1 **else** S2;	t_{max}(E) + max(t_{max}(S1), t_{max}(S2))
while (E) S;	$(N+1) \times t_{max}$(E) + $N \times t_{max}$(S)

While the pure timing-schema approach is easy to use it has a severe problem. It cannot process information about the execution characteristics of a program that spans across the borders of nested constructs. For example, for the code listed in Figure 1 the timing schema approach includes the WCET bound of line 5 N times within the inner loop and N^2 times in the whole program fragment shown. Obviously this yields a pessimistic WCET bound. The timing schema does not allow the user to express the number of executions of line 5 relative to the outer loop (i.e., across the borders of the inner loop) which would yield a WCET bound that accounts only $N(N+1)/2$ times for line 5.

A way out of the above-mentioned problem is to add *path information* (i.e., information that characterizes execution paths across construct boundaries) to program code and use more complex methods for WCET analysis. While this works fine for a WCET analysis in which the path information and execution times of program parts are fully defined and for which the complexity of the WCET analysis is of minor concern,

```
for (i=1; i<=N; i++)          loop count: N
{
  for (j=1; j<=i; j++)        loop bound: N
  {
    /* calc. */     executions in outer loop: (N+1)N/2
  }
}
```

Figure 1. Piece of Code and Path Information

it is not suited for a WCET analysis of reusable and portable code. The latter analysis has to be kept simple, because one cannot expect that the means for a complex analysis are available at the location where the final WCET computation takes place. Section 3.2 will introduce path information that both helps to keep the pessimism within bounds and is still suited for a simple, on-target WCET analysis.

3 WCET Analysis for Reusable Portable Code

WCET Analysis for reusable and portable code differs from traditional WCET analysis in that not all necessary parameters for WCET analysis (e.g., the timing of instructions on the target hardware) are available at the software developer's site. Therefore the software developer cannot provide a concrete WCET bound with the delivered code. Instead, the developer provides an abstract representation of the execution time of the code that can be easily evaluated when the missing parameters are known. The entire WCET-analysis is therefore split into two parts, see Figure 2.

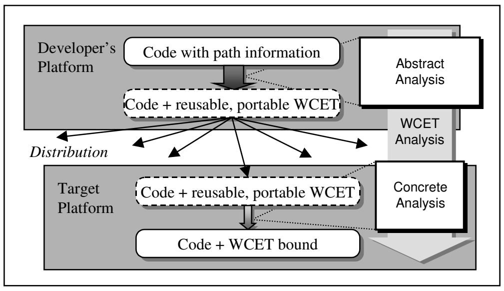

Figure 2. Two-Step WCET Analysis

The first step of the analysis is done at the developer's place. The software developer writes the code to be reused or ported, annotates it with path information, translates it into a distribution format, and generates the abstract representation of the WCET of the code. The latter step is called *abstract WCET analysis*.

Together with the distributable code the abstract WCET information is delivered to the end users (an end user could be both a human, e.g., a programmer that uses the code, or a computer system). The end user instantiates the abstract WCET information with the missing parameters and computes the final, concrete WCET bound. This step is called *concrete WCET analysis*. As mentioned before, the resources at the end user's site may be very limited and complex analysis tools may not available. The analysis must therefore allow for the concrete analysis to be simple.

3.1 Reusability and Portability

It has been mentioned that reusability and portability make it impossible that the software developer performs the complete WCET analysis. In the following we explore the effects of reusability and portability on WCET analysis in more detail.

Reusable code is (re)used in different application programs. Because the exact application context in which the code will be used is unknown at development time, the software developer cannot know the exact paths on which the software will execute. The software developer can only provide *abstract path information* for WCET analysis. This abstract path information has to be instantiated with concrete values by the end user.

Portable code is ported to and executed on different hardware platforms. For portable code the timing parameters (e.g., processor speed) of the target hardware are not exactly known to the software developer. Therefore, the duration of the actions the software performs is unknown. The WCET analysis of the software developer can therefore only provide *abstract durations* of actions (instructions) of the code. Again, the end user has to instantiate these abstract information to obtain concrete execution-time bounds.

While both, reusability and portability, influence the information about the execution time that can be represented, the static control structure of the code is not affected by these properties. The control structure of the code is, of course, a determining factor for the WCET of the code. Reusable and portable WCET information therefore can be summarized to consist of

- information about the static control structure of the code,
- abstract execution-time information for the actions/instructions of the code, and

- abstract information about the paths an execution may take in the target application.

The first two items are directly derived from the analyzed code, i.e., once the specified functionality has been coded and translated into the reusable and portable format, they are exactly determined. Path information, in contrast, is meta-information about code behavior that is not so tightly coupled to the code. The designer of the abstract execution-time format therefore has the freedom to choose the path information to be supported by that format.

3.2 Path Information for Reusability and Portability

The path information supported by the abstract execution-time format is determined by two considerations:

- The path information in the abstract representation must allow for a simple calculation of the WCET by the "end user", when the concrete path information and instruction timing are known.
- The path information information must have sufficient expressive strength to yield WCET bounds that are not too pessimistic.

As a mimimum requirement the reusable, portable path information must be able to represent loop bounds. Without loop bounds, no WCET estimate can be computed at all. As for the path information to be supported on top of loop bounds, there is the following dilemma: If no path information besides loop bounds were allowed, the resulting WCET bounds would generally be too pessimistic to be of any practical relevance. If, on the other hand, full path information were allowed, then due to the global path dependencies described, the first abstract WCET analysis step could not do much useful work. As a consequence the concrete analysis would become very complex, and that conflicts with the goal to provide a simple solution for the second step of the analysis.

As a compromise between allowing full path information and none at all (besides loop bounds), our approach only allows two patterns of usage of path information (see Figure 3). The choice of these patterns may seem random. In fact, however, these patterns are part of those most frequently found in analyzed programs. Also, their use generally contributes significantly to obtaining tight WCET bounds.

Both patterns describe the execution frequency of one or more code sections relative to a specific part of program code, a so-called *scope* [13]. In Figure 3 scopes are marked by thick-lined boxes with name S.

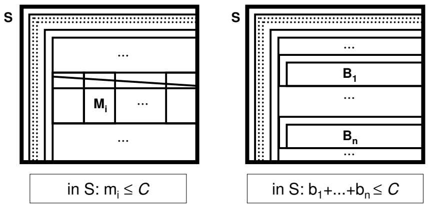

Figure 3. Allowed Path-Information Patterns

The first (left) pattern sketched in Figure 3 makes it possible to characterize the maximum number of executions of one or more locations M_i, called *markers*, within S with a constant C (m_i represents the number of executions of M_i). Within S, all M_i must be directly included in the same loop and all surrounding loops must not contain any branching statements. There is no restriction on the use of statement sequences within the loop and branching statements.

The second pattern provides for the possibility to describe an upper bound on the number of executions of a set of loops occurring in a statement sequence. This bound is additional to the individual loop bounds B_i. It expresses that the iteration number of all loops together, i.e., $\sum b_i$, is not greater than a constant C. Again, there is the restriction that none of the surrounding loops shall contain a branching statement.

3.3 Abstract WCET Analysis

So far we have introduced the basic structure of the two-step WCET analysis and the patterns of path information the user is allowed to use. The rest of this section describes the WCET analysis in more detail.

The abstract-analysis tool generates the reusable and portable execution-time information. It first reads the portable code and generates a tree data structure that represents the control structure and the abstract timing of the operations of the analyzed program. The tool then traverses the tree structure and generates the abstract execution-time formula: All parts of the tree except scopes are translated into timing-schema expressions (e.g., an *if-then-else* yields an expression $c_1 + \max(c2, c3)$, where $c1$, $c2$, and $c3$ are representations of the abstract execution times of the condition, *then* and *else* branches, respectively). For parts of the tree that represent a scope and the markers generating the abstract timing information is more complex.

3.3.1 WCET Analysis for Scopes

Scopes typically contain a number of nested loops, and not all loops iterate the same (=maximum) number of times on each invocation. As shown before, the knowledge about the maximum number of executions of a part of code within the nested loops of a scope is coded by a marker. The first step of the scope analysis computes the maximum numbers of executions for all loop bodies within the scope. These maximum execution counts are computed as the minimum of the products of the bounds of the surrounding loops and the maximum number of executions described by markers. Because at least part of the path information (loop bounds, marker limits) to be manipulated may be unknown at the time of this analysis, the computation is done symbolically. The tool uses the algorithm sketched in Figure 4. To analyze a scope a reference to the outer loop and a count of 1 are passed to the function.

```
L.count : max number of L's iterations in scope
L.bound : upper bound on loop iterations

AdjustBounds (Loop_description L, integer C)
{
  L.count = C * L.bound;
  if is innermost_loop (L) then
    for all markers m set m' = min(m, L.count)
    for all subpaths without markers loop
      Collapse them into a single alternative
      with m'=L.count;
    end loop;
    L.count = min (L.count , sum of all m');
  else
    for SL in subloops of (L) loop
      AdjustBounds(SL,L.count );
      L.count = min(L.count, SL.count);
    end loop;
  end if;
}
```

Figure 4. Adaption of Loop Counts and Markers

Knowing the *ExecutionCounts* of the loops, the execution-time terms for the loops are added to the abstract WCET. The branching statement that contains the markers is excluded from this loop analysis. The abstract execution-time of the branching statement is computed separately.

3.3.2 Analysis of the Branching Statements with Markers

Each branch is characterized by a pair (m'_i, c_i), where m'_i is the maximum execution count (adapted marker) of the path and c_i the abstract representation of its execution time.

Now, let us derive the abstract WCET of the branching statement. For the sake of simplicity let us first assume that the worst-case scenario allows all branches to execute the maximum number of times, m'_i. In this case the WCET of the branching statements is the sum of the contributions of all branches,

$$\sum m'_i c_i.$$

In general, the assumption that *all* branches execute with maximum frequency in the worst-case scenario does not hold. This is due to the fact that the maximum number of executions of the branching statement is not just bounded by the sum of the markers, but also by the product of the maximum iteration counts of all loops into which it is enclosed (e.g., the statements enclosed in two nested loops, each of which has a loop bound of 10 simply cannot be executed more than $10 \times 10 = 100$ times). Let us call the iteration limit imposed by the loop bounds and the markers M (Note that M equals the value *L.count* as computed by the algorithm shown in Figure 4).

Given an iteration limit M and N branches with maximum execution count m'_i and execution time c_i ($1 \leq i \leq N$), and assuming for the moment that $c_i \geq c_j$ for $i < j$, the WCET of the branching statement is constructed as follows: Initialize the WCET of the branching statement to $m'_1 c_1$ (according to the algorithm from Figure 4, $m'_1 \leq M$). Subsequently add as many iterations as possible of the branches with execution times c_2, then c_3, c_4, etc. until a total of M iterations have been reached. Once M iterations have been reached, add zero iterations for all remaining branches. This calculation yields the following WCET formula for the branching statement within its scope:

$$\sum_{j=1}^{N} \min\left(m'_j, \max\left(M - \sum_{k<j} m'_k, 0\right)\right) c_j.$$

In the construction of this formula we assumed that we are able to order the branches according to their execution times. This is, of course, not possible until we know the timing parameters of the target. The sum for the abstract WCET representation therefore has to take into account that each of the branches might be the longest, second-longest, etc. on the final target. The following formula does so by "generating" all permutations Π of (m'_i, c_i)-pairs. It computes the WCET of the branching statement as the execution-time value of that permutation p that yields the maximum sum.

$$\max_{p \in \Pi\{(m'_i, c_i)\}} \sum_{j=1}^{N} \min\left(m'_{pj}, \max\left(M - \sum_{k<j} m'_{pk}, 0\right)\right) c_{pj}$$

This formula represents the abstract WCET computation term of the branching statement.

The result of the steps described is a complete WCET formula for a piece of code. The structure of the formula reflects the structure of the code and the abstract path information. Constants describe the known path information, and symbolic expressions characterize abstract information about paths and target-specific execution times of basic operations.

3.3.3 Abstract Instruction-Execution Times

We do not propose a particular format for the abstract instruction-timing information in this paper. The choice of this format depends on the particular portable code format used and the desired degree of portability. For example, a portable execution-time representation for a piece of Java byte codes may be a vector that represents how often each of the available byte codes occurs in the code, see [1]. On the other hand, if the portable format is machine code that is to be ported between computers with equal processors but memory of different speeds, then the abstract execution time may be coded by the number of computational CPU cycles and the number of memory accesses.

3.4 Concrete WCET Analysis

The concrete WCET analysis instantiates the abstract values of the reusable portable WCET information with concrete values and computes the WCET bound. This requires that the concrete analysis provides concrete values for all path-information parameters as well as for the abstract information characterizing the timing of the basic operations on the target.

Once the concrete information is available the concrete analysis is simple. It replaces all abstract values in the WCET representation by the concrete values and evaluates the resulting formulas to obtain the concrete WCET bound. The formulas only contain the following operators: assignment, (,), $+$, $-$, $*$, min, and max.

4 An Example

The following example demonstrates our method of calculating execution times. Figure 5 shows the code for Bubble Sort. The possible execution paths are characterized as functions of the path parameter n in the form of two loop bounds (lines 6 and 8), and a marker (line 12) in scope S (lines 5 to 19). Note that the preferable position for the marker would actually be before the *if* statement, i.e., between lines 9 and 10. In order to give an example with a marker in a branching statement, however, we chose to put the marker in its current position.

```
1  void bubble(int a[])          path parameter:  n
2  {
3    int i, j, t;
4
5    begin scope S
6    for(i=SIZE-1; i>0; i--)    loop count:  n - 1
7    {
8      for(j=1; j<=i; j++)      loop bound:  n - 1
9      {
10       if (a[j-1] > a[j])
11       {
12         execution bound in S: n * (n - 1)/2
13         t = a[j-1];
14         a[j-1] = a[j];
15         a[j] = t;
16       }
17     }
18   }
19   end scope S
20 }
```

Figure 5. Code with Abstract Path Information

We now assume that the code is compiled into a portable execution format (e.g., Java byte code). In parallel the abstract execution-time analysis builds a data structure that represents the corresponding control structure, the abstract timing, and the abstract path information.

In the example the abstract execution-time information for sequential pieces of code is coded in vectors with four components. Once the concrete execution times for these components are known, the multiplication of the abstract and the concrete vector yields the concrete execution time for each code sequence.

Figure 6 sketches the data structure used by the abstract-WCET tool. The data structure represents the static program structure in the form of a tree similar to a syntax tree. There are nodes for the two loops, for the *if*-statement and its branches. The *else* branch represents the actions to be taken in case the condition of the *if* evaluates to false.

Figure 6 shows the data structures after the algorithm for the adjustment of loop bounds and markers had been run. The *maxcount* data of the loops hold the maximum number the respective loop bodies can be executed and m' stores the actual maximum each of the branches can be executed.

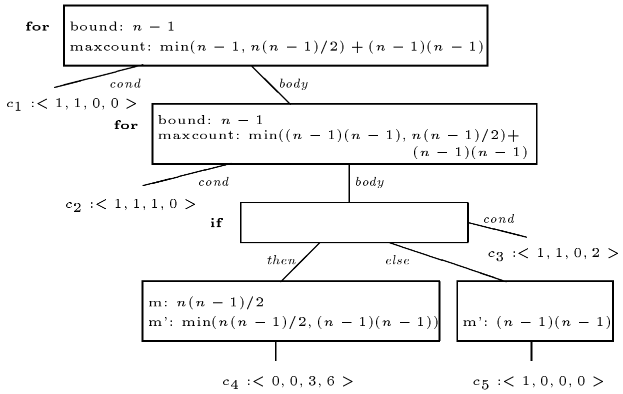

Figure 6. Representation of Abstract Timing after Loop-Bound and Marker Adjustment

The abstract timing information for the sorting program is shown in Figure 7. The variables `t1` and `t2` store the maxcounts for the loops, `m1'` and `m2'` store the marker limits generated by the adjustment algorithm. In the following abstract execution-time formula $c1$ to $c5$ represent the abstract execution-time vectors. The symbolic names have been chosen to increase readability. In the real abstract-timing formula these symbols are replaced by the vectors.

```
t1   := min(n−1, n(n−1)/2+(n−1)(n−1))
t2   := min((n−1)(n−1), n(n−1)/2+(n−1)(n−1))
m1'  := min(n(n−1)/2, (n−1)(n−1))
m2'  := (n−1)(n−1)

wcet = (1+t1)*c1*c + (t1+t2)*c2*c + t2*c3*c +
       max(min(m1', max(t2−0, 0))*c4*c +
           min(m2', max(t2−m1', 0))*c5*c,
           min(m2', max(t2−0, 0))*c5*c +
           min(m1', max(t2−m2', 0))*c4*c)
```

Figure 7. Abstract Exec.-Time Information

Once the target application and the target system are known, the abstract execution-time information is evaluated to yield the concrete WCET bound. If, for example, the target application has to sort $n = 10$ elements and the concrete execution-time vector for the target is $< 1, 1, 2, 4 >$, then the concrete analysis yields 2576 time units, see Figure 8.

5 Experimental Results

We evaluated our approach for three sample programs. The main focus of this evaluation was on the quality of WCET bounds given the restrictions on the use of path information of our approach. To this end

$n := 10, c :=< 1, 1, 2, 4 >$

`t1 = 9, t2 = 81, m1' = 45, m2' = 81`

$$\begin{aligned} \texttt{wcet} &= (1+9)*2 + (9+81)*4 + 81*10 + \\ &\quad \max(\min(45, \max(81, 0))*30 + \min(81, \max(36, 0))*1, \\ &\qquad \min(81, \max(81, 0))*1 + \min(45, \max(0, 0))*30) = \\ &= 20 + 360 + 810 + \max(1386, 81) = 2576 \end{aligned}$$

Figure 8. WCET-Calculation for the Target

we compared the WCET bounds computed with our approach, the real WCETs, and the WCET bounds computed by the pure timing-schema approach.

The three functions evaluated were C implementations of Bubble Sort and Merge Sort for seven elements and a Fast Fourier Transformation (FFT) implementation for an array of 512 elements. Bubble Sort and Merge Sort were compiled for the Motorola 68000 processor family, the concrete execution-time analysis was done for the MC68000 processor with zero wait-state memory. For the analysis of the FFT function we used the same code and timing assumptions as in [2].

Table 2. WCETs and Computed WCET Bounds

Prog.	*WCET*	*TS*	*RPT*	$\frac{RPT}{WCET}$	$\frac{TS}{RPT}$
bubble	4.3	7.0	4.3	1.00	1.63
merge	6.7	33.8	9.1	1.36	3.71
fft	2.3	589.8	3.6	1.57	163.83

Table 2 shows the results of our experiments. The numbers are in thousands of CPU cycles for Bubble and Merge Sort and in thousands of time units for FFT. The $WCET$ column shows the real WCETs of the programs, the TS column the WCET bounds computed with the simple, recursive timing schema, and the RPT column the WCET bounds computed with the here-presented approach for reusable portable timing analysis. The final two columns present the ratios $RPT/WCET$ and TS/RPT, giving a direct comparison of the results of the WCETs and the computed bounds.

The results suggest that the approach for reusable and portable WCET analysis performs quite well. From the $RPT/WCET$ column we see that it allowed us to compute the exact WCET for Bubble Sort. For Merge Sort and FFT the RPT analysis allowed us to describe only part of the path information that would be needed to compute the WCET. Still the WCET bounds stayed within 160% of the WCET. The RPT analysis outperformed the timing-schema approach in all cases, see last column. For FFT the WCET bound computed with the TS analysis is even two orders of magnitude worse than the result of the RPT analysis.

6 Conclusion

This paper presented a WCET-analysis approach for reusable portable code. The goal of this approach is to compute high-quality execution-time information that can be reused and ported together with the code. Clearly, not all information about the execution context and the target system is available when the portable WCET information is computed. The approach presented here, nevertheless, allows users to compute abstract execution-time information that can be shipped with the code.

The presented WCET analysis builds on two steps: The first step is the main step. It is performed by the software developer to compute abstract WCET information that is reusable and portable. The second step is then only a minor step. It uses the reusable portable WCET information to compute the final WCET bound when the missing parameters are known.

Naturally the focus of this paper was on the first step of the WCET analysis, the abstract WCET analysis. The paper showed how the path information for WCET analysis has to be restricted in order to keep the second step simple and thus acceptable. Also it presented the algorithms that are used to compute the abstract WCET information, which uses only a few simple operators and can thus be ported easily. The evaluation of the WCET method with sample programs demonstrated the fitness of our approach. The pessimism of the WCET bounds computed with our method stayed within 60% for the tested programs. A comparison with the results of the simpler timing-schema approach (which also allows the coding of reusable and portable WCET information) demonstrated that the additional path information allowed by our approach significantly improves the result of the analysis. In one case the WCET bound computed with the approach presented in this paper was two orders of magnitude smaller than the bound obtained with the timing schema.

References

[1] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, Dec. 2000.

[2] G. Bernat and A. Burns. An Approach to Symbolic Worst-Case Execution Time Analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*, pages 45–50, Palma, Spain, May 2000.

[3] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 18(2/3):249–274, May 2000.

[4] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174, Orlando, Florida, USA, Nov. 2000.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[6] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, Dec. 1996.

[7] S. Lim, Y. H. Bea, G. T. Jang, B. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for risc processors. In *Proc. 15th Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[8] S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19th Real-Time Systems Symposium*, pages 334–345, Dec. 1998.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[10] F. Mueller. Generalizing timing predictions to set-associative caches. In *Proc. Euromicro Workshop on Real-Time Systems*, pages 64–71, Toledo, Spain, June 1997.

[11] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[12] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis (Guest Editorial). *Real-Time Systems*, 18(2/3):115–127, May 2000.

[13] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[14] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.