Institute of Software Technology
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Fachstudie Nr. 142

Comparison of WCET Tools

Wolfgang Fellger, Sebastian Gepperth, Felix Krause

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Erhard Plödereder
Supervisor:	
Commenced:	September 1, 2011
Completed:	November 21, 2011
CR-Classification:	C.4, D.2.4

Contents

1 **Introduction** 3
1.1 Basics 3
1.2 Goals of this study 3
1.3 Participating tools 4
1.3.1 aiT 4
1.3.2 Bound-T 4
1.3.3 METAMOC 4
1.3.4 OTAWA 4
1.3.5 TuBound 4
1.3.6 WCA 4
1.4 System configuration 4

2 **How we evaluate** 5
2.1 Usability 5
2.2 Analysis Features 5
2.3 Annotation Capabilities 7
2.4 Papabench WCET results 7

3 **Test details** 8
3.1 aiT 8
3.1.1 Usability 8
3.1.2 Analysis Features 9
3.1.3 Annotation Capabilities 10
3.1.4 Papabench WCET results 10
3.2 Bound-T 13
3.2.1 Usability 13
3.2.2 Analysis Features 14
3.2.3 Annotation Capabilities 15
3.2.4 PapaBench WCET results 16
3.3 METAMOC 17
3.3.1 Usability 17
3.3.2 Analysis Features 19
3.3.3 Annotation Capabilities 19
3.3.4 Papabench WCET results 19

3.4 OTAWA 20
3.4.1 Usability 20
3.4.2 Analysis Features 21
3.4.3 Annotation Capabilities 22
3.4.4 Papabench WCET results 22
3.5 TuBound 23
3.5.1 Usability 24
3.6 WCA 24
3.6.1 Usability 24
3.6.2 Analysis Features 26
3.6.3 Annotation Capabilities 27
3.6.4 Papabench WCET results 27

4 Comparisons and results 28
4.1 Feature Comparison 28
4.2 Detailed WCET results 29
4.3 Usability 31
4.4 Conclusion 31

5 Acknowledgements 32

A Microbenchmarks 33
A.1 Constant Loops 33
A.2 Infinite Loops 33
A.3 Branches 35
A.4 Calling Contexts 40
A.5 Jumptables 41
A.6 Recursion 43

1 Introduction

1.1 Basics

Safety-critical software is commonly designed for embedded systems and, more importantly, the requirements contain hard deadlines for how fast the system must respond to certain situations; it is "real-time software". This means, *not finishing in time is as useless as not finishing at all* and must be counted as failure.

Of course, speaking of execution times as requirement introduces a very close interaction with the actual hardware chosen for the project. Suddenly, the correctness of the software depends on the choice of hardware – the execution speed *on the particular CPU* has always to be kept in mind. However, over-dimensioning the CPU is not a solution: The faster CPU does not only cost more money, but is likely less reliable as well.

It is also clear that an engineer can not simply be "confident" that the code runs in time – neither does it suffice if measurements show that the code *typically* makes the deadline.

Thus, what we need as input for all further calculations is a *guaranteed* upper bound on the worst case execution time of a piece of code, typically a "task". Unfortunately, this boundary can not be measured, because the input that would trigger the worst case is typically unknown. The general approach is to derive this number by static analysis on the compiled object code. Of course, while this estimate *must* be leaned to the safe side, it *should* be as low as possible. In this study we will have a look at some of the available tools.

1.2 Goals of this study

There are two major aspects when calculating a WCET boundary: First, statically inferring upper bounds of the number of loop and function executions; second, modelling the actual target architecture as closely as possible to arrive at a tight boundary in machine cycles. These aspects are almost completely independent; in fact, some tools split them into two separate binaries.

Our attempt in this study is evaluating and comparing a range of tools that is as broad as possible; for this reason, we will focus on the first aspect. The target architecture is mostly a means to get as many tools as possible on comparable grounds; therefore our choice fell on the comparably simple and straight-forward ARM7 architecture. Special 'tricks' such as emulating cache hits and memory access time will not play a role in our benchmarks, though we briefly list such capabilities in the introductory section of the corresponding tool.

Instead, we will focus on the tools' abilities to statically analyse the program. Most contenders do this from the compiled ARM binary; however, there is also one tool (TuBound) which works directly on the source code, as well as the relatively odd-ball WCA which works on Java source. Another aspect of this study is how well tools handle situations that they can *not* automatically resolve. We will have a look at how time-consuming it is to provide assertions, and of course general tool stability and polishedness.

After evaluating the quality of their statical analysis, the tools working on ARM binaries will then be put to a 'classic' test using the PapaBench benchmark.

1.3 Participating tools

1.3.1 aiT

One of the two commercial attenders, the aiT suite is provided by AbsInt.

Website: `http://www.absint.com/ait`

1.3.2 Bound-T

The other commercially available tool in the contest, provided by Tidorum Ltd. from Finland.

Website: `http://www.bound-t.com`

1.3.3 METAMOC

METAMOC is a relatively new tool which focuses on simulating ARM hardware; it does not do any loop boundary detection.

Website: `http://metamoc.dk`

1.3.4 OTAWA

OTAWA is a suite of the two tools oRange and oipet, developed by the University of Toulouse in France.

Website: `http://www.otawa.fr`

1.3.5 TuBound

This tool, developed at the TU Wien, works directly on source code, with the help of a source-to-source compiler.

Website: `http://costa.tuwien.ac.at/tubound.html`

1.3.6 WCA

WCA is the oddball of this study. It does not work on ARM binaries or even C source, but on an imaginary hardware implementation of Java. While there is a Java port of the PapaBench, its results are completely incomparable to the C version.

Due to the simple nature of our microbenchmarks, however, it can partake in the first part of this study.

Website: `http://www.jopwiki.com/WCET_Analysis`

1.4 System configuration

We chose the ARM7 platform for this comparison, as this allows the largest subset of tools available on the market.

To arrive at comparable numbers, we will use a plain ARM7 TDMI configuration without caching and memory waits.

All our benchmark binaries are compiled using arm-elf-gcc 4.6.1 with optimizations disabled and we made sure to use the exact same binaries for all tests.

More precisely, all sources listed in appendix 1 are compiled using the following command line and then subjected to all tools (with the exception of METAMOC (3.3) and WCA (3.6), see their respective chapters).

```
arm-elf-gcc FILE.c -g -mcpu=arm7 -o FILE.elf
```

GCC itself is compiled with this configuration:

```
./configure --prefix=/usr --target=arm-elf --disable-nls
--enable-languages=c,c++ --enable-multilib --enable-interwork
--with-local-prefix=/usr/lib/cross-arm --with-as=/usr/bin/arm-elf-as
--with-ld=/usr/bin/arm-elf-ld --with-newlib --with-float=soft
--host=x86_64-unknown-linux-gnu --build=x86_64-unknown-linux-gnu
```

Using this configuration, we ran into an unexpected difficulty regarding the standard library. The ARM7 does not have a floating point unit, but the PapaBench uses floating point data types, which means that the operations are compiled as a function call. Two of these functions, `__aeabi_fmul` and `__aeabi_dmul`, proved to be particularly difficult to analyse automatically. As far as this can be expressed (see the respective chapters for details), we provide the following assertions to the tools:

- All loops in `__aeabi_fmul` repeat exactly once
- `__aeabi_dmul` takes 80 cycles.

This is, of course, an oversimplification. Thankfully, we are not actually required to arrive at a valid WCET result for PapaBench, but are only interested in comparable results.

2 How we evaluate

Every tool will be briefly introduced in their respective chapters and is then evaluated in regard to usability, analysis features and annotation capabilities. Finally results from analysing the PapaBench benchmark with the individual tool will be presented.

2.1 Usability

In terms of usability, the tools are examined in the aspects of installation procedure, general usage of the tool, documentation, stability and error communication. Following the explanation of the usability, the encountered obstacles are described.

2.2 Analysis Features

The main purpose of our microbenchmarks is to allow us to draw conclusions about the analysis capabilities of the corresponding tools. The mapping from microbenchmark results to features is as follows:

Static Loop Bound Detection

With this test we check whether the tool can derive loop bounds from a loop that can easily be statically bounded, and whether it arrives at the correct result.

Relevant test cases:

CLOOP1 (Listing 1: constant_loop1.c): Simplest constant loop

CLOOP2 (Listing 2: constant_loop2.c): Constant loop with step size != 1

Infinite Loop Detection

This tests how the tool responds to infinite loops. Of particular interest is the clarity of the error message. Of course, in no case should the tool crash in these simple tests.

Relevant test cases:

INFINITE1 (Listing 3: infinite_loop1.c): while(1); loop

INFINITE2 (Listing 4: infinite_loop2.c): never ending arithmetic loop

Arithmetic Loop Bound Detection

This tests how far the tools can go when deriving loop bounds. A simple loop which performs integer division by two until zero is reached.

Test case:

INFINITE3 (Listing 5: infinite_loop3.c): finite loop using constant division and a potential, but unused, break

Branch conditions within a function

Tests whether the tool can follow a static condition variable within a function call to exclude branches that are never taken.

Relevant test cases:

BRANCH1 (Listing 6: branches1.c): Very simple if(1) test.

BRANCH2 (Listing 7: branches2.c): A static expression is used as condition.

BRANCH4 (Listing 9: branches4.c): Tests basic support for switch statement

Following branch conditions over function boundaries

The logical continuation of the previous feature is tracking branch conditions over function boundaries.

Relevant test cases:

BRANCH3 (Listing 8: branches3.c): One function boundary

BRANCH5 (Listing 10: branches5.c): Two function boundaries

Calling Contexts

These benchmarks test the capabilities of the tool to differentiate loop boundaries by calling contexts.

Relevant test cases:

MLOOP1 (Listing 11: simple_method_loop1.c): A function is called with two different constant parameters.

MLOOP2 (Listing 12: simple_method_loop2.c): Same as previous, but the parameters are static expressions.

Handling of dynamic function calls (Jumptables)

Selecting and calling a function from an array of function pointers is a common pattern, but also very hard to statically analyse. However, a tool could support the user in creating the necessary annotations by collecting possible candidates.

We will distinguish two levels of support:

- Solve the most simple situations automatically (first test)
- If an annotation is required, gather the possible candidates for the call and present them in some form (second test)

Relevant test cases:

JUMP1 (Listing 13: jumptables1.c): using a constant index.

JUMP2 (Listing 14: jumptables2.c): a real dynamic call, based on user input.

Recursion handling

While recursion is a powerful programming tool, it introduces the risk of stack overflow, is difficult to statically bound, and thus often outright forbidden in real-time applications.

This test is mostly to see whether the tool detects the recursion and how it reports it. Ideally, it would allow to enter a maximum recursion depth. As expected, no tool was able to bound this automatically.

Relevant test cases:

RECURSION (Listing 15: recursion.c): Recursive Fibonacci implementation.

2.3 Annotation Capabilities

In this section we will discuss the "mightiness" of the assertion language in detail.

This is roughly based on the feature test above (as we expect some tests to require user input). We will explain how convenient assertions can be created, how jump tables are handled, whether or not certain code paths can be "cut off" by declaring an explicit time for it, and the languages capabilities in referring to code blocks/locations.

2.4 Papabench WCET results

We test a total of ten entry points from the PapaBench for ARM, denoted in table 1.

There are only two loops within these entry points, namely in `send_data_to_autopilot_task` and `servo_transmit`. The final WCET results will use 10 for both loops, however we will note if a tool is able to calculate the bound by itself.

Module	Entry point	Shorthand
Autopilot	altitude_control_task	AA
	climb_control_task	AC
	link_fbw_send	AL
	stabilisation_task	AS
	__vector_5	AV
Fly-By-Wire (FBW)	check_failsafe_task	FCF
	check_mega128_values_task	FCM
	send_data_to_autopilot_task	FSD
	servo_transmit	FST
	test_ppm_task	FT

Table 1: Tested Papabench entry points

3 Test details

3.1 aiT

aiT is part of the *AbsInt Advanced Analyzer* software distribution by AbsInt Angewandte Informatik GmbH. aiT is available for many different processor architectures, including ARM. It incorporates loop boundary calculation and WCET analysis in one application which is operated with a graphical user interface. aiT is available for Windows- and Linux-based systems.

3.1.1 Usability

Installation.

The Linux version of aiT provides a simple interactive shell script for installation which worked without any problems.

Concept.

aiT is an integrated, graphical tool for WCET computation. It guides the user through the process of executing a WCET analysis from the specification of input binaries to the viewing of computation results. Amongst other features, aiT includes panes for specifying input files, processor and memory configuration, an editor with syntax highlighting for annotation files, a disassembly viewer, an interactive call graph viewer and some inspector views on the input (like e.g. a list of symbols).

Familiarisation.

aiT is very easy and straightforward to use. While it does provide a lot of configuration options, one can get to a first WCET computation result very quickly. Advanced configuration through annotations can be added later.

Assertion creation.

Assertion files can be written with aiT's integrated editor, which features syntax highlighting for the annotations and a wizard to create new annotations. Annotations can be applied per computation target, globally for the whole project or as part of a configuration profile.

Documentation.

aiT includes a PDF manual as well as a quick reference for its annotation syntax. We found especially the latter one very useful.

Stability and error communication.

aiT runs stable and did not crash once while we worked with it. It reports errors and warnings while computing a WCET target as part of the log output, highlighted with red and orange background respectively. It is possible to filter the log for warnings and errors.

3.1.2 Analysis Features

Loop Bound Detection

aiT was able to calculate all static loop bounds correctly without further configuration. It also gave informative feedback when encountering the infinite loop tests telling the user which loop is or may be unbounded. However, aiT was not able to process the non-obvious finite loop INFINITE3, which requires arithmetic analysis.

Branch conditions and calling contexts

Both branch conditions determinable within a function and branch conditions dependent on a determinable calling context were processed correctly by aiT. It recognized the branches that will never be called and excluded them from the WCET calculation. Considering calling contexts, aiT was able to distinguish calls to a function between their different calling contexts.

Handling of dynamic function calls (Jumptables)

aiT was able to resolve the computed call in JUMP1 as it is non-ambiguous. In JUMP2. It tells the user that there are unresolved computed calls and continues to calculate a WCET without information about possibly called functions. It clearly states in the output that the result may not be a WCET. It is possible to annotate the dynamic function call and tell aiT which functions can possibly be called by it.

Recursion handling

aiT simply reports that the problem RECURSION1 is unbounded. However, it is possible to annotate the maximum recursion depth.

3.1.3 Annotation Capabilities

aiT provides a plain-text, human-readable annotation syntax. Items in the binary, such as functions, loops or branches, can be referenced by their name (if they have one), their address, or their relative location in another structure, like e.g. the third loop in a function with a certain name.

The following information can be annotated:

- Targets of computed calls and branches
- Compiler
- Clock rate of target processor
- Loop bounds
- Properties of memory areas
- Register values
- Recursion
- Properties of routines and calls that should not be analysed
- Infeasible code

It is also possible to specify additional output information, like the calls to a certain function.

3.1.4 Papabench WCET results

We wrote the following global annotations (which correspond to those used with Bound-T):

```
loop "__mulsf3" + 1 loop exactly 1;
loop "__mulsf3" + 2 loop exactly 1;

snippet "__aeabi_dmul" is not analyzed and takes max 80 cycles
        and does not violate callingconventions;
```

Additionally to those global annotations, we needed additional annotations for some entry points.

Module Fly-By-Wire:

Without annotation the function `__floatsidf`, AbsInt would output an error saying that it encountered an unresolved branch in some - not in all - calling contexts. To get a decent result, we annotated the WCET Bound-T calculated of this function, which is 74 cycles:

```
snippet "__floatsidf" is not analyzed and takes max 74 cycles
        and does not violate callingconventions;
```

Curiously, the calculated WCET of another function `__adddf3` changed when we added this annotation. The call graph indicated that both functions share some anonymous code. As these functions are functions from the standard library and not the PapaBench

itself, we did not have the time and resources to track down the real cause of this issue. A comparison of the call graph before and after this annotation are depicted below.

The affected entry points are: FCM, FSD and FTP.

For FST, AbsInt would report that the problem is unbounded. We manually specified the boundary of 10 (Bound-T was able to deduce this automatically):

```
loop "servo_transmit" + 1 loop exactly 10;
```

Module Autopilot:

The entry points we needed an additional annotation for in this module were AC and AS. The problem here was similar to the problem with `__floatsidf` in the other module. This time, the problematic function was `__extendsfdf2`. Again, we had a call to `__adddf3`, which changed its WCET after we annotated `__extendsfdf2` (only in AC - AS does not call `__adddf3`).

```
snippet "__extendsfdf2" is not analyzed and takes exactly 79 cycles
        and does not violate callingconventions ;
```

With this, we arrived at the results shown in table 2.

Papabench results (cycles)

Autopilot					Fly-By-Wire				
AA	AC	AL	AS	AV	FCF	FCM	FSD	FST	FT
399	1694	58	1611	99	1080	1659	729	873	3745

Table 2: Papabench results of AbsInt

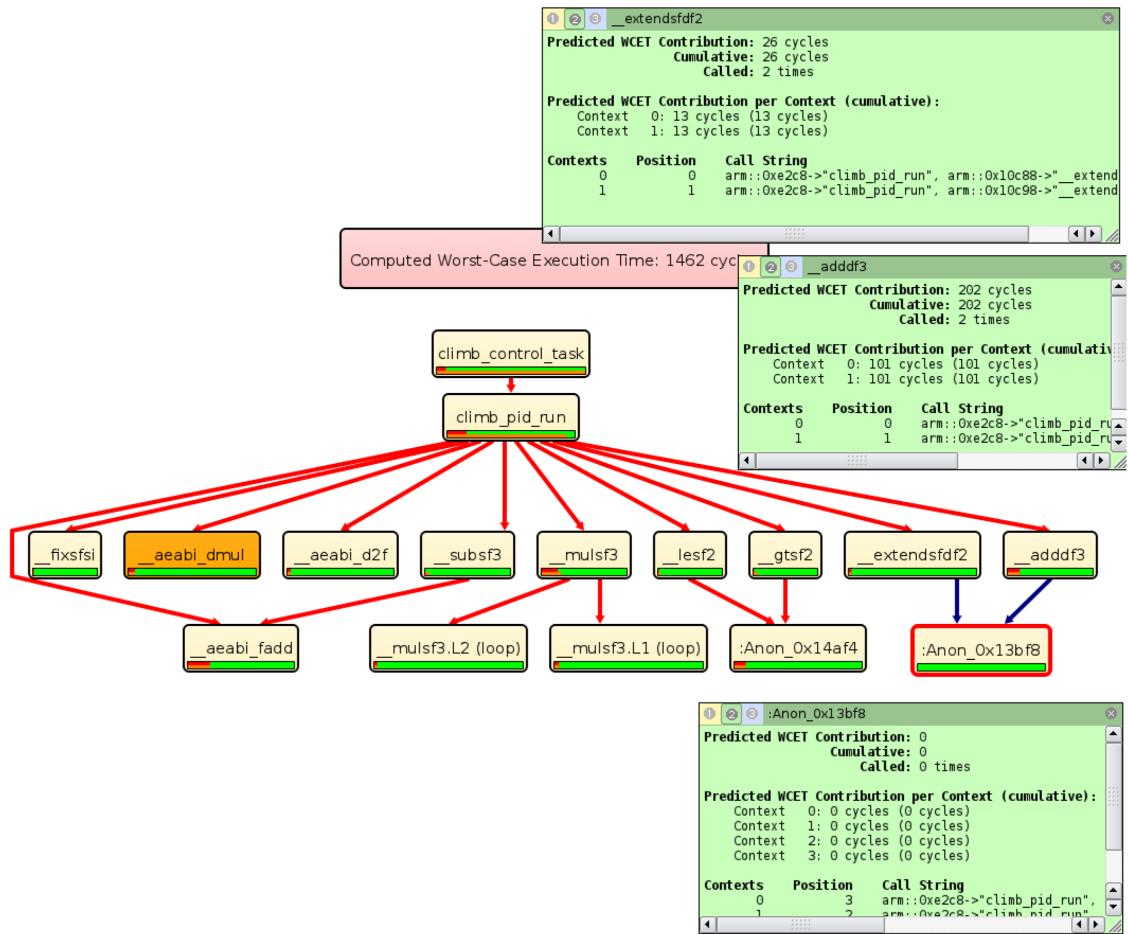


Figure 1: Screenshot of the AbsInt call graph before annotating `__extendsfdf2`

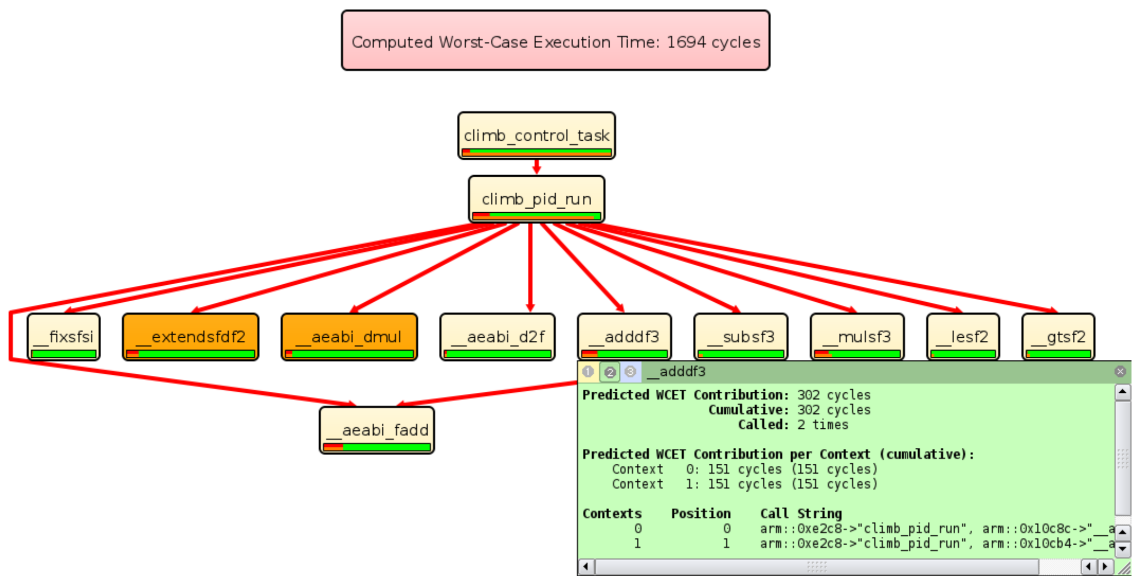


Figure 2: Screenshot of the AbsInt call graph after annotating `__extendsfdf2`

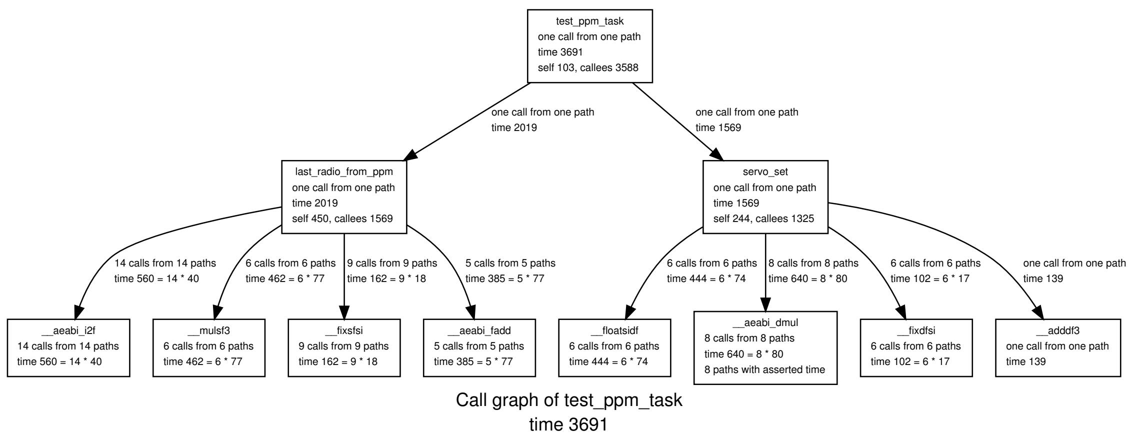


Figure 3: A typical call graph produced by Bound-T

3.2 Bound-T

Bound-T is the other commercial tool in this contest and to our knowledge the only contender written in Ada. Compared to AbsInt, it is much less polished, lacking an installer or a graphical frontend. Features and documentation are comparable, though, and Bound-T seems to have a slight advantage when automatically estimating loop bounds.

3.2.1 Usability

Installation.

Two archives with the main and auxiliary binaries have to be downloaded and unpacked. The only non-trivial part is that adjustments to the search path are required before it runs correctly. Failure to do so leads to the tool crashing mid-analysis. Since the tool explicitly needs the exact versions of the auxiliary binaries anyway, it would have been easier if it would just look in a fixed path relative to the main binary.

Concept.

Bound-T is a 'unix style' command line utility, meaning that its output is meant to be post-processed by other filters. For example, for the call graph visualization, a GraphViz script is created. As a consequence, every aspect is easily scriptable, partly making up for the lack of convenience functionality in the tool itself.

Familiarisation.

Bound-T is straight-forward to use, and the web site provides a good example demonstrating the most important features. Beyond that, there is extensive documentation available for both the tool itself as well as the assertion language.

Time for running the benchmark.

Bound-T was able to automatically able to infer most of our loop boundaries. After using a shell macro to contain the relevant parameters, all microbenchmarks could be quickly

processed.

For the Papabench benchmark, Bound-T had no problems at all, apart from difficulties with two arithmetic functions from the linked-in standard library (see below).

Assertion creation.

Not surprisingly, assertions are given via external text files. This of course means that there is no syntax support at all, and while the basic syntax is easy to learn, Bound-T's assertion language is very powerful. For a casual user, looking up details in the reference manual is virtually required.

Documentation.

Extensive documentation is available both for the program as well as its assertion language. The website also provides a simple "tutorial" which demonstrates all basic functionality in a step-by-step fashion.

Stability and error communication.

On one of our test machines, the tool tended to quit with a non-descript CALCULATOR_ERRORs when tabular output was requested. The exact same operation worked flawlessly on another machine. We worked directly with the maintainer, Niklas Holsti, in an attempt to resolve this rather mysterious issue, but were unable to pinpoint it.

Apart from that, one would expect a command line tool to terminate on problems anyway, and the messages usually pinpoint the location precisely that needs the user's attention.

Export functions.

Since graph drawing works via means of GraphViz, Bound-T's graph generation is excellently scriptable and accessible for further usage. Any other output is text and can be trivially captured anyway.

In practice, choosing from the multitude of available graph drawing options commonly required a glance into the manual and could probably profit from a graphical frontend.

3.2.2 Analysis Features

Loop Bound Detection

Bound-T was able to determine most loop boundaries in the micro benchmark automatically, detected the infinite loops, and has a way to ignore them for the purpose of immortal tasks.

However, infinite_loop3.c, which is actually finite, could not be bounded. According to the manual, only addition, subtraction and constant multiplication is supported for deriving loop bounds, while this loop uses constant division.

Bound-Ts did not succeed in bounding the two arithmetic functions from the standard library, `__aeabi_fmul` and `__aeabi_dmul`, but could bound every loop in the PapaBench, contrary to AbsInt.

Following branch conditions within a function

It appears that branch conditions can be followed within a function only.

The functionally identical branch2.c (listing 7) and branch3.c (listing 8) thus had vastly different WCET bounds – for the second, the longer, but actually unused branch was assumed.

This turned out to be Bound-T's main disadvantage when compared to AbsInt and reduces the usefulness of its calling context implementation.

Calling Contexts

Calling contexts are supported and mostly useful for loop bounds. Conditional execution seems only to be taken into account "downwards" the call graph, not if the condition bases on the result of another function call.

Handling of dynamic function calls (Jumptables)

Dynamic calls are supported, but can only be resolved automatically in specific cases, such as a switch statement. In any other case, the user is asked to provide a list of possible functions in the annotation. No effort is made to build a list of candidates automatically, even if they could be determined statically (see listing 13).

Recursion handling

Recursion is detected and reported with an appropriate error message. This aborts the analysis, and there is no possibility to resolve the situation, e.g. by specifying a maximum recursion depth manually.

3.2.3 Annotation Capabilities

Bound-T has a very extensive assertion syntax. Particularly noteworthy is the ability to specify code locations: Apart form line numbers, one can also specify blocks such as loops relative to each other and based on the used variables, so the assertion can 'survive' simple modifications in the program.

3.2.4 PapaBench WCET results

The following assertions were used for PapaBench:

```
subprogram "__aeabi_dmul"
        time 80 cycles;
end subprogram;

subprogram "__mulsf3"
        all loops repeat 1 time; end loop;
end subprogram;
```

Please note that none of the loop bounds in PapaBench itself had to be specified, but were correctly deduced automatically.

Papabench results (cycles)

Autopilot					Fly-By-Wire				
AA	AC	AL	AS	AV	FCF	FCM	FSD	FST	FT
383	1637	56	1570	97	1597	1640	705	801	3691

Table 3: Papabench results of Bound-T

Comparison with AbsInt

The tools are pretty much on par, with Bound-T typically a few cycles lower than AbsInt; probably due to slight differences in the memory model. We will have a closer look at three instances that were of particular interest: AC (Autopilot: `climb_control_task`), FCF (Fly-By-Wire: `check_failsafe_task`) and FCM (Fly-By-Wire: `check_mega128_values_task`).

climb_control_task

It is to be noted that in `climb_control_task`, Bound-T manages to automatically arrive at a bound for `__extendsfdf2` and `__adddf3` at 79 / 139 cyles, while AbsInt generates an "Anon" block which it is unable to analyse.

Since this is deep in the standard library and probably hand-optimized assembly, we did not press the issue further.

When the length of `__extendsfdf2` is given to AbsInt, the warning for `__adddf3` vanishes as well and it yields a result of 1694 cycles.

check_failsafe_task and check_mega128_values_task

As can be seen in figure 4, the call graphs of these two functions are almost the same. In Bound-T, the results for all subprograms are also the same, in particular 74 cycles for `__floatsidf` and 139 cycles for `__adddf3`.

AbsInt seems to be able to make more use of calling contexts, reducing the call to `__floatsidf` to 5 cycles for `check_failsafe_task` (it is, however, unable to bound it

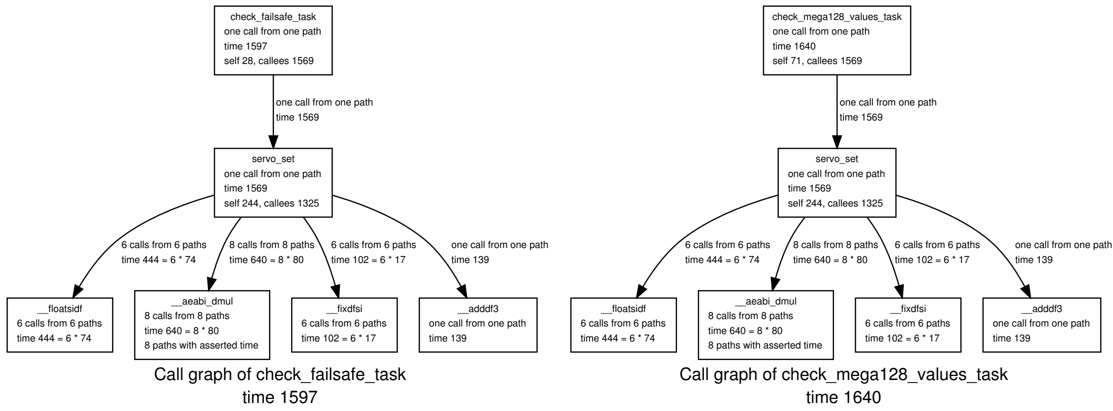


Figure 4: Call graphs of `check_failsafe_task` and `check_mega128_values_task`

when it actually executes in `check_mega128_values_task`). At six calls, this makes for a large advantage of AbsInt.

3.3 METAMOC

METAMOC is a relatively new contender with a novel approach: Instead of statically deriving properties from the binary or source, the CPU itself is modelled, and then handed to a model checker. This means that a call graph in the classical sense is never constructed, and non-determinism can be used for unknown dynamic properties.

Unfortunately, the tool itself clearly shows its experimental nature by an utter lack of polishedness. In its current state, it is extremely hard for outsiders to work with.

Additionally, METAMOC is focused on precisely emulating the ARM9 architecture. It does not do any loop boundary detection, which means it cannot take part in the first part of this study.

As a result, we are probably not giving it the honour it deserves; an eye should definitely be kept on further development of this tool.

3.3.1 Usability

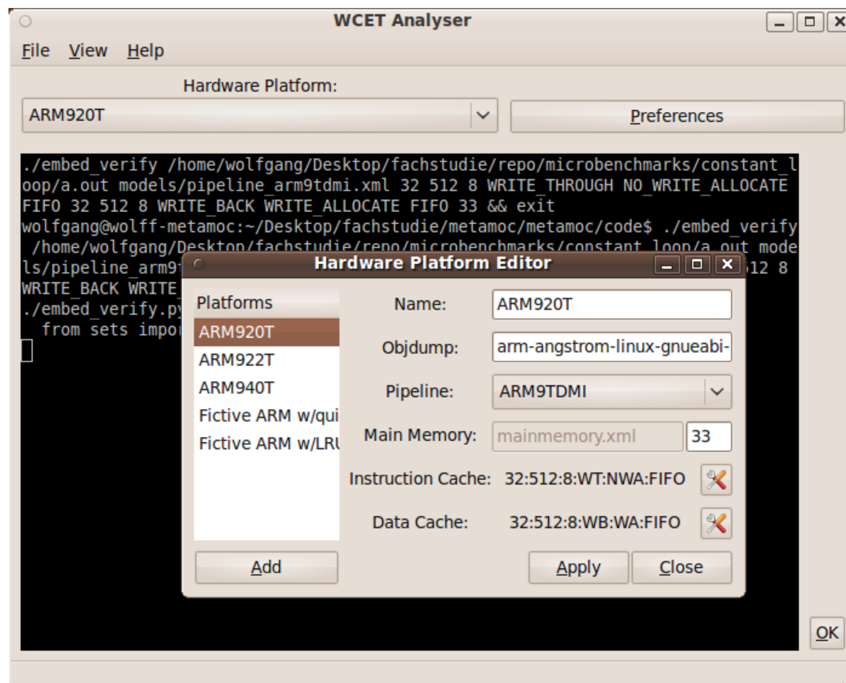
Installation.

METAMOC is only available as source, so has to be compiled before it can be used. It relies on a now deprecated version of Python to build, so we decided to set up a virtual machine for it. We tried to make it work with arm-elf toolchain instead of the default arm-angstrom-linux-gnueabi toolchain, but did not succeed. We resorted to recompiling the benchmark using arm-angstrom-linux-gnueabi-gcc, losing comparability.

The value analysis reproducibly locked up during our tests, which could only be resolved using the command line tools and personal communication with the maintainer.

Concept.

METAMOC parses the output of GNU `objdump` and generates a system model, which is then handed to the graphical model checker UPPAAL. An actual WCET value can be retrieved by issuing the appropriate query.



The METAMOC GUI frontend. Unfortunately, we never saw it work.

Figure 5: METAMOC

The GUI is supposed to automate and hide the usage of both tools, but we did not get this to work properly.

This led to the following very unfortunate workflow:

- Run command-line tool with appropriate parameters to avoid the crashing value analysis
- Run UPPAAL on the generated model file
- Navigate model and find places where missing loop bounds can be entered
- Manually enter a specific query to retrieve WCET value.

Needless to say, this was very confusing and time-consuming for someone unfamiliar with the tools and environment like us.

Assertion creation.

Assertions can either be given after model creation within the model checker UPPAAL, or directly in the source code using a specially-formed comment.

Both methods are, in our opinion, not very usable: Source code annotations obviously require repeated recompilations and are thus impossible in situations where the binary is fixed; annotating in the UPPAAL GUI requires that the user can complete the above workflow himself, which includes typing in a specific query string.

A mode that generates the model file, interactively asks for the loop bounds to use, and then continues to calculate the WCET would have been more helpful in our opinion.

Documentation.

One of the largest problems for outsiders like us is the lack of documentation. There is virtually no documentation available at all, apart from the original thesis the program is based on.

Stability.

As noted above, the value analysis tended to lock up, consuming full CPU power and ever more memory.

Furthermore, several functions from the standard library linked into the PapaBench used opcodes that were not supported and aborted the analysis. To make matters worse, METAMOC relies on UPPAAL to generate the call graph, meaning it will always process every function in the file, so there is no way to resolve this except editing the source code.

Error communication.

Errors produced by the main program were clear enough; the locking value analysis obviously did not give any indication at all.

The more complex benchmarks in our test, for example the jumptables, lead to UPPAAL complaining about syntax errors in the model and were totally indecipherable for us.

3.3.2 Analysis Features

As mentioned above, loop bound detection is not implemented.

For some reason, the infinite loop in listing 3 is not even detected as loop, and the system yields a WCET time of 28 cycles (which is thus the most significant underestimation case in this study).

The recursion is also not detected and handled as loop. This may actually work in simple cases, but looks more like an oversight than a feature.

3.3.3 Annotation Capabilities

Annotations are very simple and only allow to specify loop boundaries.

Because of the above mentioned syntax errors in the generated model, we did not get to test how this is supposed to work with situations that require more complex input - our only assumption is that it is currently impossible.

3.3.4 Papabench WCET results

Since METAMOC does not perform any loop boundary detection, our hopes were with the PapaBench.

Unfortunately, this turned out to be mostly impossible as well, for the following reasons:

- As mentioned above, we can not use the same binary as for the other tools. Since annotations are given in source code, we can not even use the same same source.
- Also as mentioned above, the value analysis feature did not work at all.

- METAMOC crashes when analysing some of the standard library functions.
- All functions linked into the binary are unconditionally analysed; there is currently no way to specify another entry point than `main` anyway.

After manually trimming and rearranging AV's code into a standalone binary and changing the CPU model to the one we assume, we finally managed to get a number of 125 cycles. For comparison, Bound-T evaluated the modified code at 163 cycles, and AbsInt at 165 cycles.

The slightly lower value may be due to METAMOC simulating an ARM9 instead of ARM7, or us being unable to completely match the cache model.

3.4 OTAWA

OTAWA is actually a collection of several related tools, mainly *oipet*, *oRange* and the *OTAWA Eclipse plugin*.

3.4.1 Usability

Installation.

Both the command-line tools and the Eclipse plugin can be downloaded as binaries for Windows and Linux. The command-line tools contain a simple install script while the Eclipse plugin can simply be copied into the Eclipse dropin folder. There are no further dependencies to be resolved.

Concept.

oipet is a command-line tool for WCET calculation which takes a binary file and one or multiple entry points. oipet does not calculate loop bounds, so one can provide a *Flow Facts* file where the loop bounds of the binary are specified.

oRange is a command-line tool for determining loop bounds that works on source code. Unfortunately, it produces XML output, which is incompatible with oipet. There is no tool for converting oRange's output into a *Flow Facts* file as required by oipet, but it can be done by hand.

The *OTAWA Eclipse plugin* integrates into the Eclipse integrated development environment for C/C++. It is a graphical frontend for oipet and simplifies the specification of the flow facts. It also features a control flow graph viewer.

Familiarisation.

oRange and oipet provide a set of options which is easy to understand. The XML output of oRange is human-readable and not too complex. As we did not use Eclipse for C/C++ development before, we cannot give a detailed rating on how well it integrates into the Eclipse workflow, but this is definitely a feature worth mentioning; the other tools covered here do not integrate in an IDE.

Assertion creation.

When using the Eclipse plugin, assertions can be created on the fly. The WCET calculation queries any loop bounds and other information it needs in order to finish the calculation when it is launched. The needed fields are displayed in a tree view of the current test where the fields can be filled with the required information. It is also possible to load *Flow Facts* files that give information on loop bounds.

Documentation.

The oipet tool has some documentation on the OTAWA wiki page. The OTAWA Eclipse plugin has a few html pages documentation which were made available by Christiane Rochange for the WCET Challenge 2011. As for now, no other public documentation on the plugin is available. For oRange, there is no documentation available apart from the command line output.

Stability and error communication.

The Eclipse plugin occasionally yields a segmentation fault, causing Eclipse to crash. On one of our testing systems, it crashed reproducibly every time we wanted to open the window for choosing a processor configuration. On another system, this worked without errors. The Eclipse plugin works with the Galileo and Helios releases of Eclipse. It does not work with the current Indigo release.

The command line tools mostly worked without errors or displayed understandable error messages.

3.4.2 Analysis Features

Loop Bound Detection

oRange was able to calculate all loop bounds correctly. Notably it was the only tool which did not report that the loop in the test INFINITE3 was unbounded. Instead, it calculated the correct loop boundary of 5. When encountering infinite loops, oRange reports a WCET value of `NOCOMP`.

Branch conditions and calling contexts

oRange processed all branching tests without problems. It was able to calculate which branches will never be executed. oRange was also able to distinguish loop bounds between calling contexts of a function. But in these tests (MLOOP1 and MLOOP2) it reported that the calculated loop boundaries per calling context may not be exact, even though they are.

Dynamic function calls an recursion

The OTAWA tools do not provide support for automatically resolving dynamic function calls. However, oipet is able to take a specification of possible targets of the dynamic call

from the user. The Eclipse plugin queries the possible targets from the user in its GUI.

oRange does not support analysis of recursive calls. It even outputs a `Fatal error: exception Failure("parameter")` when analyzing recursive code. oipet handles recursion just like loops, it needs a maximum iteration value in order to process the recursion. In the Eclipse plugin, one can annotate this just like a normal loop boundary.

3.4.3 Annotation Capabilities

oipet (and the Eclipse plugin, which incorporates oipet's features) calculates a WCET based on flow facts which can be given as a *Flow Facts* file or entered in the Eclipse graphical user interface. The tool does not attempt at calculating any flow facts from the binaries.

oipet works on different hardware configurations by the means of specifying a processor and a cache model (defined in XML) at the command line. In Eclipse, this can be done with a graphical selection dialog.

While oRange's output specifies loop boundaries dependent on the calling context, oipet's *Flow Facts* input is not able to handle this information. Here, loop boundaries can only be specified globally as one number. Neither differences between calling contexts nor the information about whether the boundary is exact or an estimate can be included. The first restraint is the one that has an impact on oipet's WCET calculation possibilities, forcing it to apply the maximum loop boundary from all calling contexts to every execution of a loop.

3.4.4 Papabench WCET results

One of our microbenchmarks incidentally was more interesting than planned, and unveiled a bug in OTAWA, leading to missed calls and underestimated WCETs.

At the end of `jumptables1.c`, one can see a "divide by 10" operation. Our intention was to have something we could distinctively see in the disassembly before the very compiler-specific return from main begins. While we knew the ARM7 does not have a floating point unit, we oversaw that there is also no integer division. As a result, the compiler replaced it with a function call. We then noticed that OTAWA did not include this function call at all.

After further enquiry with the maintainers, we could gather the following:

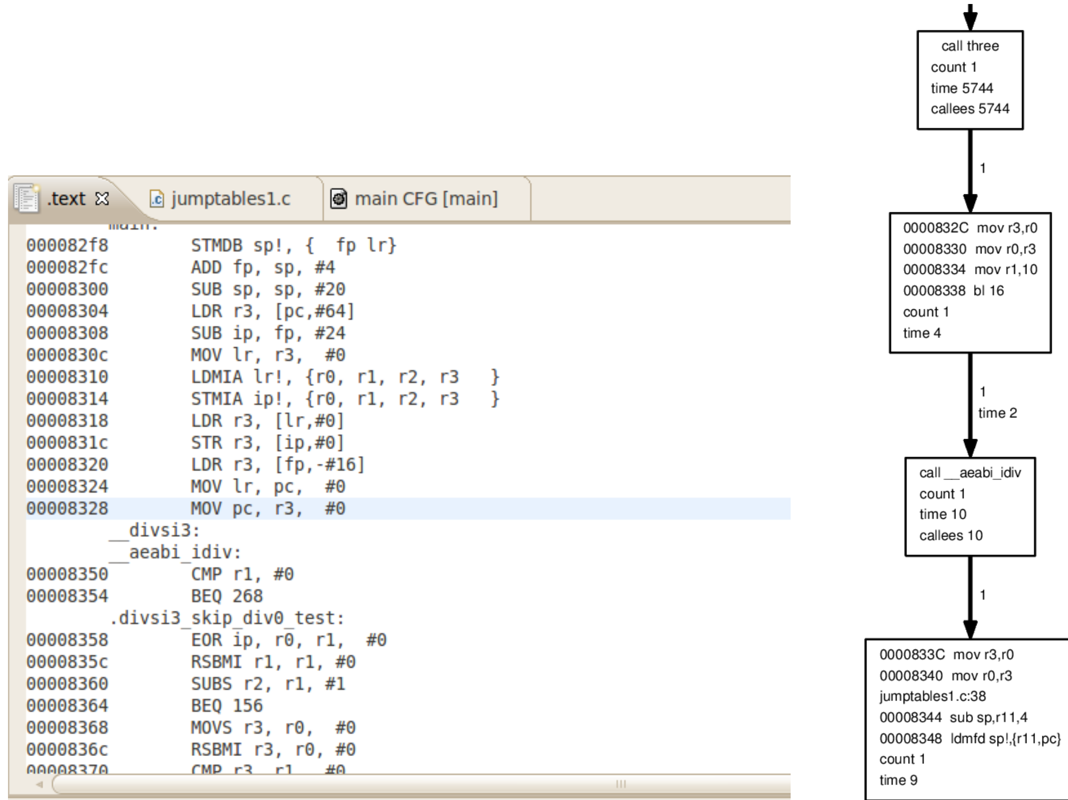
- The compiler decided to translate the dynamic call not as one of the usual `Bxx` instructions, but as direct writes to the LR (return address) and PC (instruction counter) registers.
- OTAWA failed to recognize this as a function call. At the time of writing, there is also no possibility to specify this in annotations (the maintainers assured us that they will fix this.)
- As a result, OTAWA silently ignored any instructions in `main` following this call, not even including them in the disassembly (see figure 6)

Of course, distinguishing a function call from a simple branch is tricky in the general case, and we do not blame OTAWA for that. However, what we find bothering is the fact that OTAWA behaved completely silent about this.

Since the ignored instructions were even missing from the disassembly, a user has no chance to see what is happening here unless he actively compares with another tool,

which is certainly not to be expected by an average user (even we were lucky to have stumbled upon this.) The end result is rather catastrophic: OTAWA delivers a well-hidden underestimation on the WCET.

This discovery is probably related to similar suspicions we had during the "Daimler experiment" in the WCET Challenge 2011, which indicates that this problem is not limited to the ARM platform.



The call to `three` at 0x8328 is the last instruction for `main` shown in OTAWA. For comparison, the flow and disassembly for the following instructions as produced by Bound-T.

Figure 6: Missing instructions in OTAWA

Papabench results (cycles)

Autopilot					Fly-By-Wire				
AA	AC	AL	AS	AV	FCF	FCM	FSD	FST	FT
405	1797	56	1693	97	2066	2112	895	871	4239

Table 4: Papabench results of OTAWA

3.5 TuBound

TuBound is a WCET analysis and program development toolchain currently developed at the Vienna University of Technology. It operates with source code and uses the ROSE compiler for source-to-source transformation and analysis.

3.5.1 Usability

Installation

TuBound depends on some large libraries, most prominently the ROSE compiler, which is only available as source code and needs some time to compile.

TuBound itself also has to be compiled, but despite efforts on both the TuBound team's and our side the compilation could not be completed successfully. It was therefore not possible to evaluate TuBound in this study.

3.6 WCA

WCA is the WCET analyser of JOP, the Java Optimized Processor, an implementation of the Java Virtual Machine in hardware and is maintained by the Vienna University of Technology in Austria.

3.6.1 Usability

Installation

WCA is open source and available as part of JOP(Java Optimized Processor). The source code can be obtained per git from the jop repository.

JOP depends on Altera Quartus, a closed source FPGA design software, which has to be downloaded from the Altera website and is installed with a graphical installer.

Building of JOP itself is accomplished exclusively with `make` and is very straight forward. JOP does not install itself onto the system, but stays in its source directory.

Concept

The tool is used via command line, operates on Java source code and generates HTML reports.

Familiarisation

WCA (and JOP) is controlled completely with makefiles and the `make` command, and can only be used from inside the JOP directory. Furthermore, the code to analyse has to lie in a specific location within the JOP directory tree. Usage of this tool typically consists of first running the WCET analysis with a command like this:

```
make P1=microbenchmarks P2=branches P3=branches1 java_app wcet \
 WCET_METHOD=main USE_DFA=yes WCET_OPTIONS="--wcet-preprocess"
```

This compiles the Java sources and then executes various analysis tools.

After successful completion of the analysis, a HTML report can be generated from the analysis output, again with a `make` command, which contains call graphs produced with `dot`, as shown in Figure 7.

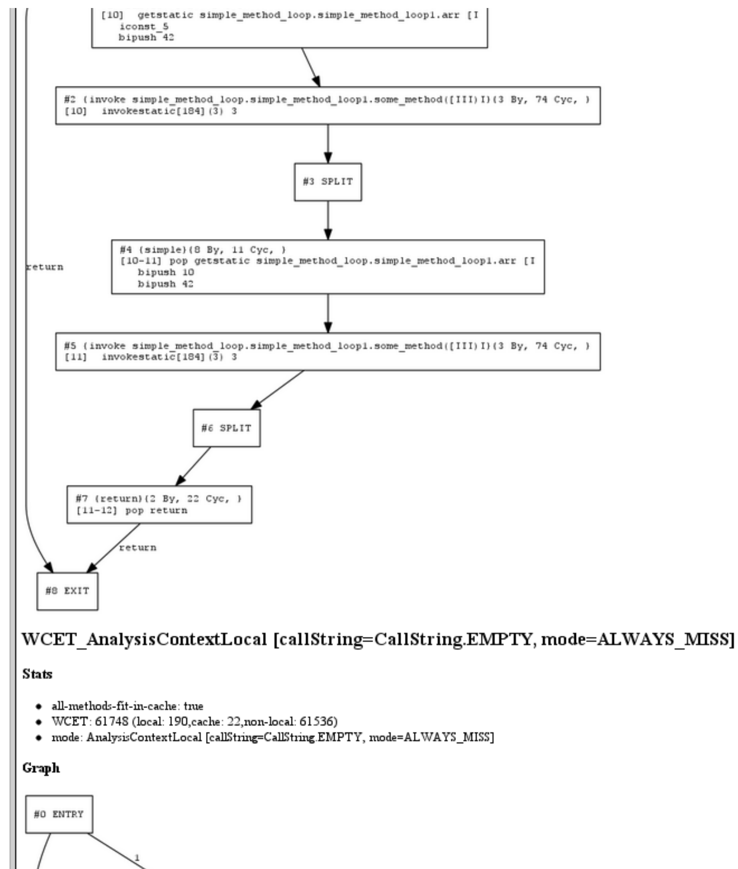


Figure 7: WCA HTML Report

Assertion creation

Loop Bound assertions can be annotated directly in the Java source code, in the same line as or directly after the loop construct.

Documentation

The Java Optimized Processor in general has a very good documentation and the WCET principles are described in detail in the handbook. As usage documentation for WCA exists only a wiki page and a text file which describes the annotation language.

Error communication

Errors are mostly in form of Java exceptions with stack trace or Java compilation errors and are not very readable.

Export functions

Instead of HTML, a CSV file can be generated.

Obstacles and Problems

Since WCA cannot be installed on the system and has to be used out of the JOP source tree, the files to analyse must be placed therein.

The generated reports are at times rather cryptic, especially for the results of the data flow analysis.

There also exists an unresolved problem with detecting a cyclic call graph while creating new objects (whithin GC.newObject), which prevented us from testing some ported microbenchmarks.

If loop bounds could not be automatically determined, a default value of 0 to 1024 is set and the computation succeeds with a warning hidden somewhere in the console output.

3.6.2 Analysis Features

The following features have been evaluated with ported versions of the C microbenchmarks.

Static Loop Bound Detection

WCA detects static loop bounds automatically without problems.

Infinite Loop Detection

The tool detects obvious infinite loops within the control flow graph (test case INFINITE1), but only communicates this in the command line tool and not in the report. It does not detect complex infinite loops (test case INFINITE2), but bounds the loop with null and approximates then with (0,1024).

Arithmetic Loop Bound Detection

WCA cannot calculate complex loop bounds as in INFINITE3 and assumes a default of 1024.

Branch conditions within a function

Branches are correctly detected and considered. The unused branch in test case BRANCHES1 is being detected early in the call graph. The call graph for BRANCHES2 is generated completely and wcet values for every branch are calculated. In the final WCET calculation the branch is correctly recognized.

Switch cases are not correctly detected and in BRANCHES4 the tool assumes a fall through starting at the first case.

Following branch conditions over function boundaries

The tool can determine branch conditions through one function call (BRANCH3), but not more (BRANCH5).

Calling Contexts

WCA was not able to track variables through method calls and use the parameters of a method call. It approximates the loop bound with the default value of 1024.

Handling of dynamic function calls (Jumptables)

The microbenchmarks for testing jumptables have been ported to Java with abstract classes and inheritance, since function pointers are not possible in Java. However, WCA does not handle object creation correctly and returns with a cyclic call graph in GC.newObject.

Recursion handling

Recursions are being detected and a cyclic call graph error is thrown.

3.6.3 Annotation Capabilities

The annotation language of WCA allows a definition of loop bounds with exact, lower and upper bounds and references to Java constants or method arguments with simple arithmetic operators. Some documented features, like method arguments, seem not to be implemented.

3.6.4 Papabench WCET results

For WCA we used the already ported jPapabench by Michal Malohlava, which is also included in the JOP distribution. The WCET results cannot be compared with those of the other tools, since both the tested software and the hardware platform are completely different to the rest.

The port is not complete and amongst other things, link_fbw_send is not implemented and the analysis of check_failsafe_task aborts with cyclic call graph in GC.newObject.

Papabench results (cycles)

Autopilot					Fly-By-Wire				
AA	AC	AL	AS	AV	FCF	FCM	FSD	FST	FT
29054	126515	(21)	156974	-	CCG	9710	11574	-	4629

Table 5: PapaBench results from WCA

4 Comparisons and results

4.1 Feature Comparison

Feature	aiT	Bound-T	METAMOC	OTAWA	WCA
Static Loop Bound Detection	✓	✓	✗	✓	✓
Arithmetic Loop Bound Detection	✗	✗[2]	✗	✓	✗
Infinite Loop Detection	✓	✓	✗[4]	✓	(✓)
Branch conditions within a function	✓	✓	✗	✓	(✓)
Cross-function branch conditions	✓	✗	✗	✓	(✓)
Calling Contexts	✓	✓	✗	✓	✗
Handling of dynamic function calls	✓	✗[1]	✗[5]	(✓)[1]	✗
Recursion handling	(✓)[3]	✗	✗[6]	(✓)[3]	✗

[1] Calling one of several targets supported, but no attempt to find candidates automatically
[2] Only addition, subtraction, and constant multiplication is supported
[3] One can provide a maximal recursion depth manually.
[4] Tool ignored the infinite loop
[5] Syntax error in resulting model
[6] Recursion interpreted as loop

Table 6: Feature Comparison Chart

Loop Bound Detection

oRange of the OTAWA suite is the unexpected winner here. It was able to solve even the complicated arithmetic loop in listing 5.

Although Bound-T has only limited support for arithmetic loop bound detection, in practice it showed to make a difference. Only two assertions were required for the entire PapaBench, while AbsInt struggled significantly more.

Calling contexts and following branch conditions

AbsInt, Bound-T and oRange of the OTAWA suite all are able to differentiate between calling contexts, e.g. can make use of the fact that a function may have wildly different execution times when called with different parameters.

In our tests, however, Bound-T was not able to follow condition variables through function call boundaries. This put it at a serious disadvantage if e.g. parts of a function are always skipped in certain calling contexts. This not only showed in the microbenchmarks, but also clearly in the case of FCF, where it is over 500 cycles behind AbsInt.

The situation is even more unlucky for the OTAWA suite: While oRange has capabilities

on par with AbsInt, the results cannot be communicated to the oipet utility; the maximum loop count must be used for all contexts (as a result, the WCET for FCF and FCM are almost the same.)

Handling of dynamic function calls (Jumptables)

It clearly shows that AbsInt has put a lot of effort into this area, and it has by a wide margin the best capabilities to resolve dynamic calls automatically or semi-automatically (e.g. by describing the structure of records containing pointers).

Bound-T and OTAWA are both able to handle branches to one of multiple targets, but require the user to manually specify the candidates.

Unfortunately, we were unable to find out whether METAMOC has capabilities in this regard.

Recursion handling

Recursion is a mostly ignored language feature in the area of real-time programming, so the basic support provided by AbsInt rounded off its image as the most feature-complete tool in this contest.

METAMOC seems to have a more accidental support for this, treating recursion as a loop.

4.2 Detailed WCET results

Entry Point	aiT	Bound-T	METAMOC	OTAWA	WCA
AA	399	383	-	405	29054
AC	1694	1637	-	1797	126515
AL	58	56	-	56	(21)
AS	1611	1570	-	1693	156974
AV	99	97	-	97	-
modified AV	165	163	125	-	-
FCF	1080	1597	-	2066	-
FCM	1659	1640	-	2112	9710
FSD	729	705	-	895	11574
FST	873	801	-	871	-
FT	3745	3691	-	4239	4629

Table 7: PapaBench Result Comparison

Microbenchmark	aiT	Bound-T	OTAWA[3]	WCA
CLOOP1	221	219	-	222
CLOOP2	88	86	-	89
INFINITE1	∞	∞	-	99
INFINITE2	∞	∞	-	99
INFINITE3	∞	∞	-	2174129
BRANCH1	41	39	-	129
BRANCH2	51	49	-	154
BRANCH3	60	875	-	253
BRANCH4	48	46	-	17439
BRANCH5	114	2028	-	35189
MLOOP1	686	684	-	61748
MLOOP2	695	693	-	61773
JUMP1	5874	5840[1]	-	-
JUMP2	9591[2]	9652[2]	-	-
RECURSION	-	-	-	-

[1] After giving the assertion "`three` is called"
[2] After giving the assertion "One of `one` to `five` is called"
[3] We did not calculate WCETs of the microbenchmarks with OTAWA, because most of the features we tested for are implemented in oRange, which is not directly able to hand over its results to oipet. For OTAWAs WCET features, refer to the Papabench results.

Table 8: Microbenchmark Result Comparison

4.3 Usability

The tools we tested have quite different concepts for user interaction: AbsInt is a project-oriented configuration tool which provides the user with a rich set of parameter input and configuration. All configuration except the annotations is stored in one file and thus is easy to migrate and archive for later reference.

OTAWA has taken some first steps into the same direction, but has still a long way to go in terms of stability and usability. Besides the occasional and sometimes reproducable crashes of the Eclipse plugin, it currently fails to save the created tasks and configuration of the OTAWA project view into the Eclipse project, so the user has to configure everything again when he closes and re-opens Eclipse.

Bound-T takes a completely different approach as it a command line tool which is designed to be used in user-defined scripts. Considering that the user of a WCET tool is usually a professional, this seems to be a decent approach, and as there is plenty of documentation available, its core is not really harder to use than the GUI-based tools. In particular, being scriptable opens it for applications such as automatically running in a build script, and tasks such as "export a graph for all of these entry points" become possible.

WCA and METAMOC, and in particular TuBound, are all currently too cryptic and hard to set up to be really usable without extensive personal support from the maintainers.

A clear winner in terms of additional functionality is AbsInt, which provides tools like a syntax-aware editor for annotations including a wizard for easily adding annotation lines, and an interactive callgraph viewer. The other tools have static output which may be viewed, but cannot be interacted upon.

OTAWA's concept of integrating into a development environment also seems to be a good idea, but the implementation is evidently not stable enough to be used in an actual project. However, it should be mentioned that OTAWA's approach to calculate a WCET with a GUI utilizes more direct user interaction than AbsInt does: While AbsInt does tell the user when annotations or configuration parameters are missing, it is up to the user to navigate to the right panel in the GUI and do the needed configuration. OTAWA, on the other hand, asks the user on the fly for loop bounds or branch destinations. While this approach may be simpler for the user, it also has drawbacks, like not being able to set a loop boundary dependent on the calling context. A GUI which provides such features in an OTAWA way might easily be a lot more complex.

4.4 Conclusion

It comes as little surprise that the two commercial solutions come out as the "winners" of this study focusing on usability and analysis features. oRange came up as a surprising match in capabilities, sometimes even outperforming both AbsInt and Bound-T. Unfortunately, it is held back by the complete lack of integration with the oipet utility - incompatible not only in syntax, but in features, which is odd for two tools that are supposed to make up a suite. We can only recommend the OTAWA maintainers to further work on bringing both tools up to par.

We were also able to track down a case of WCET underestimation in oipet, caused by misjudging a function call for a simple branch and thus declaring the program terminated too early. Doing so without any visible warning to the user is in our opinion unacceptable for professional usage, and we hope that this particular issue will be fixed soon.

METAMOC sounds like a promising approach, but is currently still too immature to really

stand the test.

Since WCA is only working with the JOP enviroment and only possesses some basic features and a simple annotation syntax, it is of little use in analysing C programms and can therefore not be compared with the rest of the evaluated tools.

TuBound shows an interesting concept of a combined analysis and compilation toolchain acting directly on source code, but could unfortunately not be examined in detail.

5 Acknowledgements

We would like to thank the following people and organisations for granting us licences for their products and supporting us in using them:

Simon Wegener and Martin Sicks
AbsInt Angewandte Informatik GmbH
D-66123 Saarbrücken
Germany

Christine Rochange and Hugues Cassé
Institut de recherche en informatique de Toulouse
Université Paul Sabatier 3
118 Route de Narbonne
F-31062 Toulouse Cedex 4
France

Jakob Zwirchmayr
Technische Universität Wien
Institut für Computersprachen E-185
Argentinierstr. 8/4/E-185.1
A-1040 Vienna
Austria

Niklas Holsti
Tidorum Ltd
Tiirasaarentie 32
FI-00200 Helsinki
Finland

Mads Christian Olesen
Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg East
Denmark

Martin Schoeberl
JOP.design Java Processor Systems
Strausseng. 2-10/2/55
A-1050 Vienna
Austria

A Microbenchmarks

In this section, all microbenchmarks we used to determine the capabilities of every tool under test are listed.

A.1 Constant Loops

These are the basic benchmarks checking whether loop bound detection is done at all. The second test would unveil an implementation that just supports very basic patterns.

```
int main(void) {
        int sum = 0;
        int i;
        for (i=0; i<10; i++)
                sum++;
        return sum;
}
```

Listing 1: constant-loop1.c

Expected result: Loop bound = 10

```
int main(void) {
        int sum = 0;
        int i;
        for (i=4; i<13; i+=3)
                sum++;
        return sum;
}
```

Listing 2: constant-loop2.c

Expected result: Loop bound = 3

A.2 Infinite Loops

With these tests we analyse how an infinite loop is handled. The third test actually is a finite loop and is used to test whether the tool recognized this fact (most tools did not).

```
/* Obvious infinite loop. */

int main(void) {
        return no_return();
}

int no_return(void) {
        while (1);
        return 1;
}
```

Listing 3: infinite-loop1.c

Expected result: Infinite loop detected

```c
/* Non-obvious infinite loop. Tools should at least not crash. */

int main(void) {
	return no_return();
}

int no_return(void) {
	int x = 21;
	while (x > 0) {
		x /= 2;
		x++;
		if (x > 42) break;
	}
	return x;
}
```

Listing 4: infinite-loop2.c

Expected result: Infinite loop detected

```c
/* Non-obvious finite loop. Tools should return a maximum loop count of
    5. */

int main(void) {
	return no_return();
}

int no_return(void) {
	int x = 21;
	while (x > 0) {
		x /= 2;
		if (x > 42) break;
	}
	return x;
}
```

Listing 5: infinite-loop3.c

Expected result: Loop bound = 5

A.3 Branches

The *branches* tests include conditional branches which are never executed and thus should not affect the WCET calculation. We can determine which branches are considered to be executed by the tool by looking at the boundaries the tools give for the loops in the code.

```c
/* Simple branches test. "is_never_called" should have 0 executions. */

int main(void) {
        int i;
        if(1){
                i = test_passed();
                i++;
                return i;
        }
        else {
                i = is_never_called();
                i++;
                i /= 2;
                return i;
        }
}

int test_passed(void) {
        return 1;
}

int is_never_called(void) {
        return 42;
}
```

Listing 6: branches1.c

Expected result: One call to test_passed, no call to is_never_called

```c
/* Branch test 2: Test whether tool can 'follow' static condition
   variable.
  int_loop should have 0 executions. */

int main(void) {
        int x = 42;
        if (x < 10)
                x = 1;
        else
                x = 0;

        if(x)
                return int_loop(42);
        else
                return test_passed();
}

int test_passed(void) {
        return 1;
}

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}
```

Listing 7: branches2.c

Expected result: One call to test_passed, no call to int_loop

```c
/* Check whether tool can follow static expression through a function
   call.
   Functionally equivalent to branches2, int_loop should not be called.
       */

int main(void) {
        if(if_smaller_10(42))
                return int_loop(42);
        else
                return test_passed();
}

int test_passed(void) {
        return 1;
}

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}

int if_smaller_10(int x) {
        if(x < 10)
                return 1;
        else
                return 0;
}
```

Listing 8: branches3.c

Expected result: One call to test_passed, no call to int_loop

```c
/* Test whether tool can evaluate switch statically.
   int_loop should not be called. */

int main(void) {
        int i = 42;
        int x = 0;
        i /= 2;
        i -= 15;
        switch (i) {
                case 1: x = int_loop(10);
                case 2: x += int_loop(20);
                case 3: x += int_loop(30);
                case 4: x += int_loop(10);
                case 5: x += int_loop(10);
                default: break;
        }
        return x;
}

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}
```

Listing 9: branches4.c

Expected result: No call to int_loop, small WCET

```c
/* Test whether tool can follow if expression through two function
   calls. */

int main(void) {
        if(if_smaller_10(2))
                return test_passed();
        else
                return int_loop(100);
}

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}

int test_passed(void) {
        return 1;
}

int if_smaller_10(int x) {
        if(x < 10)
                return int_loop(1);
        else
                return int_loop(0);
}
```

Listing 10: branches5.c

Expected result: One call to test_passed, exactly one call to int_loop, WCET < 200

A.4 Calling Contexts

These benchmarks test the capabilities of the tool to differentiate loop boundaries by calling contexts.

```c
int main(void) {
        int arr[] = {1,2,3,4,5,6,42,10,13,10};
        some_method(arr, 5, 42);
        return some_method(arr, 10, 42);
}

int some_method(int arr[], int search_to, int search_for) {
        int i;
        for(i=0; i<search_to; i++)
                if(arr[i] == search_for)
                        return i;
        return -1;
}
```

Listing 11: simple-method-loop1.c

Expected result: Two calls to some_method with differing WCETs reported. A loop bound of 7 for the second call deserves special mention.

```c
int main(void) {
        int arr[] = {1,2,3,4,5,6,42,10,13,10};
        int search_to = 5;
        some_method(arr, search_to, 42);
        return some_method(arr, search_to*2, 42);
}

int some_method(int arr[], int search_to, int search_for) {
        int i;
        for(i=0; i<search_to; i++)
                if(arr[i] == search_for)
                        return i;
        return -1;
}
```

Listing 12: simple-method-loop2.c

Expected result: Same as previous

A.5 Jumptables

These benchmarks test whether the tool can resolve a call to a function pointer by itself if it can be statically determined.

```
/* Simple call from a jump table. */

typedef int(*int_func)(void);

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}

int one(void) {
        return int_loop(100);
}

int two(void) {
        return int_loop(200);
}

int three(void) {
        return int_loop(300);
}

int four(void) {
        return int_loop(400);
}

int five(void) {
        return int_loop(500);
}

int main(void) {
        int_func functions[] = {*one, *two, *three, *four, *five};
        return functions[2]() / 10;
}
```

Listing 13: jumptables1.c

Expected result: Call is resolved automatically

```c
/* Non-predictable call from a jump table. */

typedef int(*int_func)(void);

int int_loop(int x) {
        int y = 0;
        while(x) {
                x--;
                y++;
        }

        return y;
}

int one(void) {
        return int_loop(100);
}

int two(void) {
        return int_loop(200);
}

int three(void) {
        return int_loop(300);
}

int four(void) {
        return int_loop(400);
}

int five(void) {
        return int_loop(500);
}

int main(int argc, char *argv[]) {
        int_func functions[] = {*one, *two, *three, *four, *five};
        return functions[argc]() / 10;
}
```

Listing 14: jumptables2.c

Expected result: A tool could analyse all possible candidates and report the highest value; however, none did this automatically.

A.6 Recursion

Here we test how the tools handles recursion.

```
int main(void){
	return fib(10);
}

int fib(int x) {
	if(x < 2)
		return x;
	return fib(x-1) + fib(x-2);
}
```

Listing 15: recursion.c

Expected result: Tool is able to calculate a WCET.

Declaration

All the work contained within this thesis,
except where otherwise acknowledged, was
solely the effort of the author. At no
stage was any collaboration entered into
with any other party.

(Wolfgang Fellger, Sebastian Gepperth, Felix Krause)