WCET CHALLENGE 2006

BOUND-T ANALYSIS OF THE BENCHMARKS

DOCUMENT TYPE:	Report
FROM:	Tidorum Ltd
TO:	WCET Challenge 2006 Steering Group
KEY WORDS:	WCET Challenge, Bound-T, analysis results

	Name	*Date*	*Signature*
PREPARED BY	Niklas Holsti	2006-11-06	
CHECKED BY			
APPROVED BY			

CONTACT niklas.holsti@tidorum.fi, +358 (0) 40 563 9186

DOCUMENT CHANGE LOG

Issue	Date	Changed sections or pages	Changes and reasons
1	2006-11-06	All	First issue, as submitted to the WCET Challenge 2006 Steering Group to present the vendor's (Tidorum's) results on the benchmarks. No results for the H8/330-IAR target (ran out of time). Some TBAs for descriptions.
2	TBD	TBD	Planned update to include the results for the H8/300-IAR target and fill in prose descriptions.

Table of Contents

1 INTRODUCTION AND OVERVIEW

1.1 The tool and the targets

The Bound-T WCET tool from Tidorum Ltd was entered in the WCET Challenge 2006 for the following target processors:

– Renesas H8/300, using the GCC and IAR compilers. However, we ran out of time and have full results only for the GCC compiler.

– SPARC V7/V8, using a GCC-derived compiler called BCC, from Gaisler Research.

To introduce the report on this work, we briefly describe Bound-T and these target processors. Further information on Bound-T is given in the User Manual (reference [1]) and the Application Notes that describe how Bound-T works with these target processor (references [2] and [3]).

The Bound-T WCET tool

Bound-T applies static analysis to a machine-code executable file, extracting call-graphs and control-flow graphs. It analyses the integer arithmetic computations to find loop counter variables (loop induction variables) and thus bounds on the number of iterations of the loops. Bound-T then uses the loop bounds as constraints in the Implicit Path Enumeration Technique (IPET) to find an upper bound on the execution time of the program.

When Bound-T cannot find (good) bounds on loops the user writes assertions in a specific form in a text file that Bound-T takes as input in addition to the executable program to be analysed. Assertions can also state other facts about the program, for example bounds on the values of significant varibles. This report will show the assertions that were used to assist the analysis of each benchmark program. We will also try to explain why they are needed.

The functionality and user interface of Bound-T is divided into a general part, independent of the target processor and described in the Bound-T User Manual [1], and a part specific to each target processor and described in the corresponding Application Note [2][3].

The Renesas H8/300 processor

The Renesas H8/300 is an 8-bit microcontroller with a 16-bit address space. Programs for such small machines are often written with special consideration for the limitations of the machine, for example the lack of 32-bit registers and the lack of floating-point hardware. More about this target in section 2.1.

The SPARC processor

The SPARC is a 32-bit RISC processor that is much more powerful than the H8/300. SPARC processors often include a hardware floating-point unit. The SPARC is better able to run programs that are written in the same high-level style as for mainframes or workstations. More about this target in section 2.2.

1.2 Analysis procedure

The teams

The WCET Challenge benchmarks were analysed separately and more or less independently by Tidorum itself and by Ms. Lili Tan from the University of Duisburg-Essen. Ms. Tan also analysed the benchmarks with the other tools entered in the Challenge.

This report describes Tidorum's analysis of the benchmarks.

The rounds

Following the suggestion from the WCET Challenge Steering Group, we made from one to four “rounds” of analysis of each program, with increasingly ambitious goals and different levels of annotation:

- Round 0: An initial quick analysis of the program structure, without trying to compute a WCET. This round is an addition by Tidorum and not in the Steering Group's suggestion. The main difference with respect to Round 1 is the use of the option *-no_arithmetic* to skip Bound-T's “arithmetic analysis” for loop bounds. The typical result of this round is the call-graph and a list of all the loops in the program. While these results are useful for planning the WCET analysis we do not report them here because they are redundant with the results from the later rounds. This round finds a WCET bound only if the program contains no loops or dynamic jumps.
- Round 1: Analysis for WCET with the minimal (standard) set of options and annotations for this target processor. For Bound-T this means the inclusion of the arithmetic analysis. This round finds a WCET bound if all loops are automatically bounded.
- Round 2: Analysis for WCET adding assertions to handle those parts of the program (eg. loops) that could not be analysed in Round 1. This round should always find a WCET. This round is omitted if Round 1 found a WCET.
- Round 3: Analysis for WCET with improved or additional assertions to sharpen the WCET bound (reduce over-estimation) compared to the possibly rough result from earlier rounds. This round is omitted if the result from earlier rounds cannot be improved with Bound-T.

The analysis computer

The host computer on which we used Bound-T is a Compaq Presario X1000 laptop with an Intel Centrino processor running at 1.4 GHz and 512 MB RAM. The cache size is not known. The operating system was Debian Linux.

Bound-T was compiled for this machine with the GNAT 3.15p compiler in the normal way with Ada run-time checks enabled including integer-overflow checks and stack-overflow checks. The GNAT compilation options were *-g -O2 -gnato -fstack-check*. For long analyses most of the computation time is spent in the Omega Calculator (program *oc*) which was compiled using *g++* with the options *-g -O2*. Although the Omega Calculator often uses between 0.5 and 1 GB of RAM, thus using virtual memory, we observed no thrashing in these runs, not even when the analysis was aborted for taking too long. Please note that the long execution time of the

Omega Calculator may be the fault of Bound-T which may make an inefficient translation of the target program into input for the Omega Calculator.

1.3 Overview

After this introductory section this report is structured as follows:

- Section 2 describes the chosen target processors, the chosen cross-compilers and the compilation options.
- Section 3 describes each benchmark program and its Bound-T analysis, including particular problems and their solutions or work-arounds. The analyses of the mathematical library routines for the three compilers are explained in dedicated subsections at the end of this section, in so far as the analysis is independent of the particular benchmark application.
- Section 4 presents a summary table of the benchmarks and the analysis results.
- Section 5 draws some conclusions for the future development of Bound-T.

The table in section 4 reports the resulting WCET bounds for the benchmark programs for the chosen target processors and cross-compilers. Please note very carefully that readers should *not*, from these WCET bounds, draw any conclusions regarding the code-generation performance of the compilers, firstly because the numbers are only upper bounds and may be more precise (less overestimated) for one compiler than another, and secondly because we used the same source-code for all compilers, while embedded software developers usually adapt the source-code to use the good features of their chosen compiler.

1.4 References

[1] Bound-T User Manual.
Tidorum Ltd., Doc. ref. TR-UM-001.

[2] Bound-T Application Note: Renesas H8/300.
Tidorum Ltd., Doc. ref. TR-AN-H8300-001.

[3] Bound-T Application Note: SPARC V7, V8, V8E.
Tidorum Ltd., Doc. ref. TR-AN-SPARC-001.

[4] H8/300 Programming Manual.
Renesas Technology Corporation, http://www.renesas.com.
Originally published by Hitachi Ltd. First edition, December 1989.

[5] H8/3297 Series Hardware Manual.
Renesas Technology Corporation, http://www.renesas.com.
Originally published by Hitachi Ltd. 3rd edition, September 1997.

[6] The SPARC Architecture Manual, Version 8.
Revision SAV080SI9308. SPARC International Inc. 535 Middlefield Road, Suite 210, Menlo Park, CA 94025.

[7] SPARC-V8 Embedded (V8E) Architecture Specification.
Version 1.0, October 23, 1996. SPARC International, 3333 Bowers Avenue, Suite 280, Santa Clara, CA 95054-2913, USA.

2 TARGET PROCESSORS AND COMPILATION

2.1 Renesas H8/300

The processor

The Renesas (formerly Hitachi) H8/300 is an 8/16-bit processor with a von Neumann architecture. For a full description see references [4][5]. The H8/300 has 16 general 8-bit registers which can also be used as eight 16-bit registers (pairs of 8-bit registers). Most instructions operate on 8-bit data; only a few addition and subtraction instructions work on 16-bit data.

In addition to the general (data) registers, there is a dedicated 16-bit Program Counter (PC) and a dedicated 16-bit Stack Pointer (SP). The PC is controlled with a conventional set of branch, jump and call instructions. The call and return instructions automatically push and pop the return address (PC after call) onto and from the stack (SP). Specific push and pop instructions can push and pop the general registers.

The timing of the H8/300 is very deterministic. Dynamically variable timing occurs mainly for memory access instructions depending on the accessed memory area: on-chip or off-chip.

The H8/300 is an obsolete processor; it has been succeeded by more powerful descendants with expanded architectures such as the H8/300H. Tidorum chose the H8/300 for the WCET Challenge because this processor is used in the Lego Mindstorms™ robotic construction kit that is often used to teach embedded and real-time programming. Tidorum has worked with Mälardalen University to implement a version of Bound-T for the H8/300 and this tool is used in real-time courses at Mälardalen.

The cross-compilers and the compilation options

Tidorum used two compilers: the GNU gcc compiler, version TBA, and the IAR H8/300 compiler, version TBA.

The longest “native” data type in the H8/300 is 16 bits and so we used compilation options that define the 'C' type `int` to be 16 bits. The 'C' type `long` is 32 bits, while `char` is 8 bits and `short` is 16 bits TBC. These choices were made without looking at the benchmark programs. During the analysis of some programs, we discovered that the programmers had assumed 32-bit `int` types, which means that overflows will (would) occur in the executables that we generated. These benchmarks should be corrected to use `long` variables where necessary.

TBA verbatim compiler options

Specific analysis problems on the H8/300 target

TBA prose text, the following are memos.

Switching between 8- and 16-bit parts of a variable.

Shifting loops for C expressions of the form i >>j, i<<j.

Floating point and other math routines in software.

2.2 SPARC V7/V8

The processor

The SPARC is a well-known architecture originally introduced by Sun Microsystems. For a full description see references [6][7]. The SPARC is a fairly typical 32-bit RISC machine with a von Neumann architecture. The instruction pipeline has two architecturally visible stages, decode and execute, and control-transfer instructions like jumps and calls are usually delayed by one instruction (the delay slot).

An unusual and distinctive SPARC feature is the *register file* and *register window* concept. The 32 general working registers, each 32 bits wide, are divided into 8 *global* registers, 8 *in* registers, 8 *local* registers and 8 *out* registers. There is one set of global registers, while the register file contains several (usually 8) sets of in, local and out registers. Each set is called a *register window* and provides access to 8 in, 8 local and 8 out registers. Only one such 24-register set is accessible at one time and is called the *current register window*. Typically, a subprogram gets its parameters from the *in* registers in the current window, uses the *local* registers in the current window for its own purposes, and passes parameters to other subprograms via the *out* registers in the current window.

The Call instruction is usually combined with the Save instruction which shifts the current window pointer by 16 registers in the register file so that those registers that were *out* registers before the shift (in the old window) are now *in* registers (in the new, current window). Thus, the caller-assigned parameter values in these registers, which the caller assigned as *out* registers, are now magically seen as *in* registers in the callee, while the callee can use 8 new *local* registers without any effect on the *local* registers in the old window (the caller's *local* registers). Conversely, returning from a subprogram usually executes a Restore instruction that shifts the current window pointer by 16 registers in the other direction. Thus, the callee can pass its results via its *in* registers, which the caller sees as *out* registers.

Several consecutive Save instructions can of course exceed the capacity of the register file, and then a trap occurs and the trap handler spills (stores) one or more register windows into main memory (usually into the stack) to create a new window for the callee. This is called the register file (or register window) *overflow*. Likewise, several consecutive Restore instructions can empty the register file – that is, the Restore should make the current window pointer point at a window that was spilled and is no longer present in the file. In this case a register file (or register window) *underflow* trap occurs and the trap handler loads one or more register windows from main memory (usually from the stack) to bring the required register window back into the register file.

Instruction timing in the SPARC is usually very deterministic. As in the H8/300, variation occurs mainly for memory accesses. Some SPARC models have floating-foint units and the execution times of the FP instructions are usually variable (data-dependent).

The first version of Bound-T for the SPARC was created specifically for the ERC32, a radiation-tolerant implemenation of the SPARC V7 architecture used in European space applications. The version of Bound-T used in the WCET Challenge is a further development that covers the SPARC V8 instruction set as used in the newer LEON2 implementation. However, the timing model is still that of the ERC32.

The later SPARC processors are normally used with on-chip caches (separate I and D caches). However, Bound-T does not include a cache analysis because caches were not used with the ERC32.

The cross-compiler and compilation options

To generate the SPARC binaries we chose the Bare C Compiler, BCC, from Gaisler Research. This is GCC adapted for the ERC32/LEON processor family. Version TBA.

In this compiler, the 'C' types `int` and `long` are 32 bits wide, `char` is 8 bits and `short` is 16 bits TBC.

The compiler options are conventional: -g -O2 TBA. We used the default target SPARC version which is V7, as on the ERC32. Note that SPARC V7 does not have instructions for integer division and multiplication; library routines are used instead.

Specific analysis problems for the SPARC target

TBA prose text, the following are memos.

Irreducible integer math routines: `.div`, `.rem` et al. These are considered “standard” assertions for this target and are thus allowed in Round 1 when the program uses such routines. Alternatively use SPARC V8 which has MUL/DIV instructions.

The execution time of SPARC floating-point unit (FPU) instructions is usually variable, depending on the values of the floating-point operands and on the kind of FPU implemented. Bound-T currently models the FPU of the ERC32 processor where the worst-case times are quite large, but on the other hand the worst-case times occur only for “denormalized” operands which are rare. For the analyses here reported we let Bound-T use the worst-case FPU execution times. As an experiment we also analysed the most FPU-intensive benchmark, the *papa_ap* program, with the Bound-T option *-fpu_typical* which assumes “typical” FPU execution time. This reduced the WCET bound from 62 753 cycles to 39 328 cycles (63% of the worst-case-FPU value).

3 BENCHMARKS

3.1 Introduction

This section dedicates one subsection to each benchmark program in the WCET Challenge 2006 and three subsections at the end to the mathematical libraries in each of our three targets.

For each benchmark program we give a short description of the program's nature and discuss the analysis problems and solutions for each of our three targets. We summarise the inputs to Bound-T in a table. The inputs are command-line options and assertions, the Bound-T term for “annotations” that support the analysis, most often by declaring the maximum number of repetitions of loops when Bound-T could not discover this on its own. The input tables have the following form, with the number of Rounds varying across the benchmarks. Note that the *Comp* column identifies both the target processor and the cross-compiler. In the *Options/assertions* column we include only those command-line options and assertions that influence the analysis, and omit those that only control the form and amount of results and supporting information that Bound-T emits.

Comp	Round	Options / assertions
gcc	1	Command-line options in Round 1 on the H8/300 / gcc target.
		Assertions in Round 1 for the H8/300 / gcc target, if any.
	2	Command-line options in Round 2 on the H8/300 / gcc target.
		Assertions in Round 2 for the H8/300 / gcc target.
iar	1	Command-line options in Round 1 on the H8/300 / IAR target.
		Assertions in Round 1 for the H8/300 / gcc target, if any.
	2	Command-line options in Round 2 on the H8/300 / IAR target.
		Assertions in Round 2 for the H8/300 / IAR target.
bcc	1	Command-line options in Round 1 on the SPARC / BCC target.
		Assertions in Round 1 for the SPARC / BCC target, if any.
	2	Command-line options Round 2 on the SPARC / BCC target.
		Assertions in Round 2 for the SPARC / BCC target.

3.2 Program adpcm

Nature of the program

This program is some kind of signal-processing application; the comments call it an implementation of the Adaptive Differential Pulse Code Modulation algorithm.

The program was originally written with floating-point computation. For the WCET benchmark, this was changed (by the WCET Challenge Steering Group) into (nonsensical) integer computation. However, the changes seem to assume a 32-bit `int` type, while our H8/300 compilers have a 16-bit `int` (with the compiler options

that we used). This means that some computations will overflow, which contradicts Bound-T's assumptions.

Analysis problems for the H8/300 GCC target

The overflow problem mentioned above makes it difficult to predict the number of iterations of some loops, in particular the argument-reduction loops in `my_sin` and the iterative approximation loop in the same function.

Analysis problems for the SPARC BCC target

In Round 1 we assert the time for the `.div` routine because it is irreducible and otherwise would not be analysable. Bound-T found bounds on all loops except the two argument-reduction loops in `my_sin` and the series-addition loop in the same function. On the SPARC, with 32-bit `int`, the computation of the argument for `my_sin` does not overflow. Still, Bound-T cannot compute the range using its arithmetic analysis because the expression for the argument to `my_cos`, `f*PI*i`, involves multiplication by the variable `i` and Presburger Arithmetic only allows multiplication by a constant. Although `i` is a loop counter and thus bounded (0 .. 2), Bound-T does not yet apply such bounds to expressions within the loop (this would require interval arithmetic analysis, not yet done in Bound-T).

Computing manually, we find that the argument `f*PI*i` to `my_cos` is in the range 0 .. 2000*3141*2 = 0 .. 12 564 000, which means that the argument to `my_sin` is in the range 1 570 - 12 564 000 .. 0 = -12 562 430 .. 0. This means that the first argument-reduction loop, `while (rad > 2*PI)`, does not execute at all, and the second argument-reduction loop, `while (rad < -2*PI)`, executes at most 1 1999 times. We can assert the range of the argument (rad) for my_sin, and Bound-T should the in principle be able to deduce these bounds on the loops. However, this does not work here, because Bound-T assumes the wrong signs for the literal (immediate operands) that are used in the loops. Thefore we must assert the loop-repetition bounds directly.

The number of iterations in these argument-reduction loops is context-dependent; using the value 1 999 for all calls to `my_sin` (through `my_cos`) is an overestimation by a factor of 2 in this program.

There is no simple reasoning for the number of iterations of the series-addition loop. In the real program, with floating-point computation, the bound could be based on numerical analysis. Here we assert the number 1000 which is found as a comment in the code at this point.

Options and assertions: adpcm

Comp	Round	Options / assertions
gcc	1	*-lego*
	2	*-lego*

		`subprogram "___mulsi3"`
		`  loop repeats 32 times; end loop;`
		`end "___mulsi3";`
		
		`subprogram "_my_sin"`
		
		`  -- The argument reduction loops. Because of overflow in the`

		`-- computation of "rad" with 16-bit integers, we assume that`
		`-- "rad" can have any value in the 16-bit range.`
		`all 2 loops that not call "_my_fabs" repeat 5 times; end loop;`
		
		`-- The series-addition loop; we use the comment that suggests`
		`-- a MAX of 1000. Again because of overflow, we don't know`
		`-- if the loop even terminates with 16-bit integers.`
		`loop that calls "_my_fabs" repeats 1000 times; end loop;`
		
		`end "_my_sin";`
		
		`subprogram "_scalel"`
		
		`-- The shifting loop. Some manual analysis of the code`
		`-- gives the following limit:`
		`loop repeats 11 times; end loop;`
		
		`end "_scalel";`

iar	1	
	2	
bcc	1	*-leon2 -via_positive*

		`subprogram ".div"`
		`time 149 cycles;`
		`end subprogram ".div";`

	2	*-leon2 -via_positive*

		`subprogram "my_sin" (variable "my_sin|rad" -12_564_430 .. 0)`
		
		`-- The first (decreasing) argument-reduction loop is`
		`-- unrepeatable based on the argument bounds above.`
		
		`-- The second (increasing) argument-reduction loop:`
		`loop that executes +"50" repeats 1999 times; end loop;`
		
		`-- The series-addition loop:`
		`loop that calls "my_fabs" repeats 1000 times; end loop;`
		
		`end "my_sin";`
		
		`subprogram ".div"`
		`time 149 cycles;`
		`end subprogram ".div";`

3.3 Program cnt

Nature of the program

The program initializes a two-dimensional integer array to random numbers and then traverses the array to count and sum separately the negative and non-negative array elements. There are a total of four loops, grouped into two loop-nests with an outer and inner loop. The counter bounds are 0 .. 9 in each loop.

The `Test` function contains a `float` variable `TotalTime` that is assigned (to dummy values) but never used. It seems that *gcc* optimized out this variable because

it does not appear in the symbol-table and the code contains no floating-point computation. Still, this variable should be removed from the source.

Analysis problems for the H8/300 GCC target

None. We ran only Round 1 for this simple program.

Analysis problems for the SPARC/BCC target

The program uses the irreducible library function `.rem`, for which we asserted a measured time.

Options and assertions: cnt

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

		`subprogram ".rem"`
		`  time 156 cycles;`
		`end subprogram ".rem";`

3.4 Program compress

Nature of the program

This is a data compression program from the SPEC95 suite.

General analysis problems

The subprogram compress has an inner loop using the label probe. This seems to be some kind of hash-table probing; without a deeper understanding of the algorithm we do not know how many times this loop can repeat, so we assume one execution of the loop body.

Analysis problems for the H8/300 GCC target

The arithmetic analysis in Round 1 was taking too long, so Round 1 was aborted and gave no result.

When we tried to assert loop bounds for Round 2 we found that several loops were difficult to “identify” in the assertion language, and we had to use the last-chance method of giving the machine-code address of the loop (which means that the assertions have to be updated when the program is relinked so that the loop addresses change).

Analysis problems for the SPARC/BCC target

The program uses the irreducible library functions `.div` and `.rem`, for which we asserted a measured time.

Round 1 with default options led to an overflow error (assertion failure) in the Omega Calculator during the arithmetic analysis of the cl_block subprogram. We used the Bound-T option *-calc_max* to reduce the limit on the magnitude of the literals passed to the Omega Calculator from 40 000 000 to 5 000 000. This avoids the overflow error, but the arithmetic analysis of the `compress` function was taking too long and was aborted.

The loop in the `writebytes` subprogram is bounded by the conjunctive condition `i < n && i < BITS`, where `n` is a parameter and `BITS` is 16. In theory Bound-T should discover the context-independent bound (16) and then not try for context-dependent bounds (`n`). However, BCC generates code that stores the results of the comparisons `i < n` and `i < BITS` as 0 or 1 in general registers and then computes the conjunction with a logical AND instruction. Bound-T's arithmetic analysis does not model AND instructions, so Bound-T does not discover this loop bound. We assert the fixed bound 16.

In the `output` subprogram Bound-T fails to bound the loop that calls `putbyte`, perhaps because the loop again has a conjunctive termination condition. We asserted the range of values of the local variable `bits` to help.

In the `cl_hash` subprogram Bound-T fails to bound the first loop because the (constant) initial value of the counter `i` is passed from the global variable `hsize` (through the parameter `hsize`) and Bound-T does not detect the static initialization of the global variable. We asserted the value of the `hsize` parameter.

Options and assertions: compress

<table>
<tr><th>Comp</th><th>Round</th><th>Options / assertions</th></tr>
<tr><td rowspan="3">gcc</td><td>1</td><td>-lego</td></tr>
<tr><td>2</td><td>-lego</td></tr>
<tr><td colspan="2">

```
subprogram "_cl_hash"
  -- Computed for HSIZE = 257.

  -- The loop with step -16:
  loop that executes "05FC" repeats 16 times; end loop;

  -- The loop with step -1:
  loop that executes "06A2" repeats 1 times; end loop;

end subprogram;

subprogram "_output"

   -- The loop for "code << r_off":
   loop that executes "0782" repeats 0 .. 7 times; end loop;

   -- The loop for "code >>= 8 -r_off:
   loop that executes "07B4" repeats 1 .. 8 times; end loop;

   -- The "putbyte" loop:
   loop that calls "_putbyte" repeats 1 .. 16 times; end loop;

   -- The loop for "maxcode = MAXCODE (n_bits)":
   loop that executes "08FA" repeats 9 .. 16 times; end loop;

end subprogram;
```

</td></tr>
</table>

		subprogram "_compress" -- The loop that sets "hshift" = local-word-10: loop that executes "0260" repeats 8 times; end loop; -- alternative: that defines address "lw10", TBC. -- The loop "while (InCnt >0)...": loop that calls "_getbyte" repeats 50 times; end loop; -- The loop "c << maxbits": loop that executes "02EC" repeats 16 times; end loop; -- The loop "c << hshift": loop that executes "0314" repeats 0 times; end loop; -- The "probe" loop: loop that executes "0392" repeats 1 time; end loop; end subprogram;
iar		
bcc	1	*-leon2 -via_positive -calc_max 5_000_000*
		subprogram ".div" time 149 cycles; end subprogram ".div"; subprogram ".rem" time 156 cycles; end subprogram ".rem";
	2	*-leon2 -via_positive -calc_max 5_000_000*
		subprogram "compress" -- The loop that sets "hshift": loop that defines "hshift" repeats 8 times; end loop; -- The loop "while (InCnt >0)...": loop that calls "getbyte" repeats 50 times; end loop; -- The "probe" loop: loop that is in loop repeats 1 time; end loop; end "compress"; subprogram "cl_hash" (variable address "p0" <= 257) end "cl_hash"; subprogram "writebytes" loop repeats 16 times; end loop; end "writebytes"; subprogram "output" variable "bits" 1 .. 16; end "output"; subprogram ".div" time 149 cycles; end subprogram ".div"; subprogram ".rem" time 156 cycles; end subprogram ".rem";

3.5 Program cover

Nature of the program

This program is evidently a benchmark for `switch/case` structures. It has three subprograms, each containing a single `for`-loop which contains a `switch-case` structure using the `for`–loop counter with a dense case numbering (0, 1, 2 ... limit), plus a `default` case. The non-default cases all contain the same statement while the default case has a different statement.

The number of non-default cases is 10, 60 and 120 respectively in the subprograms `swi10`, `swi50` and `swi120`. The number of for-loop iterations is 10, 50 and 120 respectively. We do not know if the inclusion of 10 superfluous (infeasible) non-default cases in `swi50` is intentional.

The compilers usually translate such dense `switch/case` structures into jumps through address tables, a form of register-indirect dynamic jump. In `swi10` and `swi50` the H8/300 *gcc* compiler seems to recognize that the non-default cases are identical and combines their code, reducing the `switch/case` to a simple decision between the `default` and non-`default` code. For `swi120` this does not happen, perhaps because *gcc* has a static limit on the number of cases that it can compare and combine. Thus, `swi120` is translated into a jump through an adress table.

This benchmark would be improved by placing different statements in the different case branches to prevent such unrealistic optimisation.

Analysis for the H8/300 GCC target

In Round 0, without arithmetic analysis, Bound-T is unable to resolve the dynamic jump in the s`witch/case` statement in `swi120`, so the flow-graph for this subprogram is incomplete. This is quite expected. However, Bound-T finds the path to the `default` case, so the incomplete flow-graph does contain a loop. The arithmetic analysis in Round 1 resolves the dynamic jump (creating the 120 branches of the `switch/case` statement) and finds bounds on the loops.

Analysis for the SPARC/BCC target

The BCC compiler for the SPARC does not combine code from the identical cases of the `switch/case` statements, at least not in this program. Using the arithmetic analysis Bound-T finds bounds on the `switch/case` indices and accesses the address tables accordingly to build the control-flow graphs. In `swi50` the bounds 0 .. 49 on the loop-index lets Bound-T omit the ten infeasible cases 50 .. 59.

Options and assertions: cover

No assertions were used.

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

3.6 Program crc

Nature of the program

As the name hints this benchmark demonstrates Cyclic Redundancy Check computation for octet strings. As is usual for SW CRC, a 256-element look-up table is precomputed to show the effect of a given octet on the accumulating CRC. This avoids looping over the bits of each octet.

General analysis problems

The look-up table is a static local variable (`icrctb`) in the `icrc` function. The table is initialized dynamically on the first call of `icrc`. This means that the first `icrc` call can take considerably longer than later calls.

The main function calls `icrc` twice. Thus, the initialization should be included in the first call but not in the second call. Bound-T's analysis does not discover this fact (because the exact effect of calls is not propagated into the caller's environment). Moreover, the Bound-T assertion language does not, at present, let us assert this fact. We can, however, compute a WCET bound that omits all initialization by asserting that the initialization code in `icrc` is never executed. This is our Round 3. We then manually compute the correct WCET bound, including only one initialization, as the average of the result of Round 1 (two initializations) and the raw result from Round 3 (no initializations). This is the WCET* reported in the summary table in section 4.

In our opinion this sort of dynamic initialization on first call should be avoided in real-time programs. The initialization of the look-up array should be in a separate subprogram.

Analysis for the H8/300 GCC target

Round 1 succeeds for this target without any assertions.

Analysis for the SPARC/BCC target

Round 1 fails to bound the loop in the function `icrc`; for some reason (under investigation) Bound-T does not find the loop-counter `j`. For Round 2 we asserted the loop using the maximum value of the parameter `len` which is 42. This maximum value can be found in the source-code or using the Bound-T option *-trace params* to display the computed bounds on parameter values. For Round 3 we excluded the initialization code as explained above and computed the better WCET bound manually from the results of Round 2 and Round 3.

Options and assertions: crc

Comp	Round	Options / assertions
gcc	1	*-lego*
	3	*-lego*

		`subprogram "_icrc"`
		`  all calls to "_icrc1" repeat 0 times;`
		`  end calls;`
		`end "_icrc";`

iar		
bcc	1	*-leon2 -via_positive*
	2	*-leon2 -via_positive*

		`subprogram "icrc"`
		`  loop that not calls "icrc1" repeats 42 times; end loop;`
		`end "icrc";`

	3	*-leon2 -via_positive*

		`subprogram "icrc"`
		
		`  -- Exclude the table initialization:`
		`  all calls to "icrc1" repeat 0 times; end calls;`
		
		`  -- Use maximum value of "len":`
		`  loop that not calls "icrc1" repeats 42 times; end loop;`
		
		`end "icrc";`

3.7 Program duff

Nature of the program

This program demonstrates “Duff's device”, a coding trick that unrolls a loop (duplicates the body a certain number of times and divides the iteration count by the same number) but still accomodates a number of iterations that is not a multiple of the unrolling factor. In this program, the loop is a copying loop and is unrolled by the factor 8. The drawback of this device is that it involves jumps into the loop body from outside the loop, which in 'C' can be done by `goto` statements or, as in this example, by mingling a `switch-case` structure with a `do-while` loop in an unstructured manner.

This device creates a control-flow graph that is not reducible, because the loop has multiple entry points (multiple loop heads).

Analysis problems

Bound-T can currently analyse only reducible flow-graphs, so cannot analyse the function `duffcopy` in this benchmark.

Options and assertions: duff

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

Results

Error message highlighting the irreducible flow-graph in *duffcopy*. Loop-bound and WCET for the *initialize* function which has a reducible flow-graph.

3.8 Program edn

Nature of the program

The main part of this benchmark seems to be some form of JPEG compression algorithm (Discrete Cosine Transform) .

Analysis problems for the H8/300 GCC target

The original benchmark used counters of type *long int* for all loops. However, all such loops in the benchmark are short enough for type *int*. Since Bound-T on the H8/300 analyses only 8-bit and 16-bit computations, we changed the loops to use counters of type *int* (through a *typedef count_t*).

The *main* function defines and initializes two local arrays, of 200 *short* integers each, for which the compiler generates two calls of *memcpy* to copy the initial data into the stack-allocated arrays. The *memcpy* function contains two loops, one for copying word-by-word and the other for copying octet-by-octet. The word-by-word loop is used when the source and destination addresses and the number of octets to be copied are all even numbers. Bound-T is unable to find bounds on these loops because they are terminated by a comparison of the initial and final destination pointers which does not fit Bound-T's concept of a loop counter. Bound-T is also unable to decide which of the two loops is used because the decision is based on the values of the pointers, which are computed from the stack pointer SP and Bound-T does not track the (absolute) value of SP.

The *latsynth* function contains a *for*-loop that Bound-T fails to bound because the counter's final value is negative. The model that Bound-T currently uses for H8/300 arithmetic (specifically the condition flags) assumes that loop counters are non-negative.

The *jpegdct* function contains several right-shift operations, which *gcc* translates into loops, where the amount of shift is given by a variable (*m* or *n*) that is modified in the outermost of the three *for*-loops that contains these shift operations. Bound-T fails to bound these shift-loops because it currently does not compute bounds on these loop-variant variables. Such bounds could be computed with Bound-T's arithmetic analysis because these variables are induction variables (constant initial value, constant increment on each loop iteration). However, because the amount of shift varies, the resulting WCET would be overestimated. We assert ranges on these variables and let Bound-T infer bounds on the shift-loops. Unfortunately we cannot use the C identifiers in these assertions, for two reasons: firstly, Bound-T for the H8/300 and *gcc* currently does not recognize symbols of local (automatic, stack-allocated) variables; secondly, although these variables are of type short (16 bit word) the gcc compiler is smart enough to use only the low octet of these words. We therefore write the assertions with the machine-level identifiers for these octets.

Analysis for the SPARC/BCC target

Round 1 is aborted in the arithmetic analysis of *jpegdct* by an assertion failure in the Omega Calculator. In Round 2 we disabled arithmetic analysis for this subprogram; Bound-T found bounds on all loops in other subprograms. The three nested loops in *jpegdct* have constant bounds and are thus easy to bound with assertions.

Options and assertions: edn

<table>
<tr><th>Comp</th><th>Round</th><th>Options / assertions</th></tr>
<tr><td rowspan="3">gcc</td><td>1</td><td>-lego</td></tr>
<tr><td>2</td><td>-lego
<pre>
subprogram "_memcpy"

   -- The word-by-word loop is the one chosen.
   loop that executes "0F80" repeats 200 times; end loop;

   -- The octet-by-octet loop is not chosen.
   loop that executes "0F8C" repeats 0 times; end loop;

end "_memcpy";

subprogram "_latsynth"
   loop repeats 99 times; end loop;
end "_latsynth";

subprogram "_jpegdct"

   variable address "lb57" 0 .. 3; -- m
   variable address "lb59" 13 .. 16; -- n

end "_jpegdct";

subprogram "___mulsi3"
   loop repeats 32 times; end loop;
end "___mulsi3";
</pre></td></tr>
<tr><td>3</td><td>TBA to improve the context-dependent “>>” loops.</td></tr>
<tr><td>iar</td><td></td><td></td></tr>
<tr><td rowspan="3">bcc</td><td>1</td><td>-leon2 -via_positive</td></tr>
<tr><td>2</td><td>-leon2 -via_positive</td></tr>
<tr><td></td><td><pre>
subprogram "jpegdct"
   no arithmetic;

   -- Outermost loop (for k in 1 .. 8)
   loop that contains (loop that contains loop)
      repeats 8 times;
   end loop;

   -- Middle loop (for i in 0 .. 7)
   loop that is in loop and contains loop
      repeats 8 times;
   end loop;

   -- Innermost loop (for j in 0 .. 3)
   loop that is in (loop that is in loop)
      repeats 4 times;
   end loop;
</pre></td></tr>
</table>

		end "jpegdct";

3.9 Program insertsort

Nature of the program

The *main* function applies an insertion sort algorithm to an 11-element *int* array. The first element at index zero is a sentinel, not data. Thus, there is an inner loop and an outer loop. The outer loop is written as a *while*-loop but is really a simple counted loop. The inner loop is a *while*-loop with a logical termination condition based on the order of array elements.

Analysis problems

The inner loop is not a counted loop. It has an induction variable (*j*), and this variable could be bounded by the fact that it is used as an array index, but Bound-T does not do such analysis automatically. The number of iterations in the inner loop depends on the order of array elements, but the maximum number of iterations depends also on the counter (*i*) of the outer loop ("triangular loop" problem). For Round 2 we asserted bounds on *j* which leads to an overestimated WCET ("rectangular" approximation). For Round 3 we asserted an average number of iterations of the inner loop to get a "triangular" WCET bound.

Options and assertions: insertsort

Comp	Round	Options / assertions
gcc	1	*-lego*
	2	*-lego*

		`subprogram "_main"`
		`  loop that is in (loop)`
		`  variable "_main|_j" 1 .. 10;`
		`  end loop;`
		`end "_main";`

	3	*-lego*

		`subprogram "_main"`
		`  loop that is in (loop)`
		`  repeats 4 times;`
		`  end loop;`
		`end "_main";`

iar		
bcc	1	*-leon2 -via_positive*
	2	*-leon2 -via_positive*

		`subprogram "main"`
		`  loop that is in loop`
		`  variable "main|j" 1 .. 10;`
		`  end loop;`
		`end "main";`

	3	*-leon2 -via_positive*

		subprogram "main" loop that is in loop repeats 4 times; end loop; end "main";

3.10 Program janne_complex

Nature of the program

This program seems to be a synthetic test case for complex loop computations and termination conditions.

Analysis problems

The program contains a loop-nest with complicated and irregular dependencies between the variables modified in the inner and outer loops and used in the termination conditions. Neither loop is a simple counted loop of the type that Bound-T can analyse, thus the only solution would be to assert iteration bounds for the loops. However, the loop logic is so complex that no simple reasoning can give the loop bounds; the program must be executed or simulated. Therefore, we have no results for this benchmark.

Options and assertions: janne_complex

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

Results

Error messages for unbounded loops and unbounded WCET.

3.11 Program matmult

Nature of the program

The program multiplies two square matrices in the straight-forward way by a 3-deep loop-nest. The loops are simple, rectangular (actually square) *for*-loops with static limits and steps. An initialization function contains a 2-deep loop-nest, also of this simple type.

Analysis problems

None, except the irreducible *.rem* library routine for the SPARC.

Options and assertions: matmult

Comp	Round	Options / assertions

gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

		`subprogram ".rem"`
		`  time 156 cycles;`
		`end subprogram ".rem";`

3.12 Program ndes

Nature of the program

The program seems to do some encryption or decryption, perhaps with the DES method (to judge from the name of the program). It defines two types *immense* and *great* that seem to represent very large integers, respectively 64 and 96 bits.

The C program contains a great number of loops, but all are simple counted loops of the kind that Bound-T can handle.

Analysis problems for the H8/300 GCC target

When compiled with *gcc* the program makes 14 calls to the *memcpy* function which contains two loops, and Bound-T cannot find the bounds of these loops, so it reports a total of 28 unbounded loops. The loops in *gcc*'s *memcpy* were described above in section 3.8. To bound the loops with assertions we must find the number of octets to be copied; this parameter is passed to *memcpy* in register *r2*. Examination of the machine code (with Bound-T's option *-trace decode*) suggests that the *memcpy* calls are used to copy *immense* and *great* values, meaning 8 or 12 octets per call. To verify this we executed Round 1 with the additional Bound-T option *-trace params* which makes Bound-T display the derived bounds on parameters for calls to subprograms where the parameters may help to find context-dependent loop bounds. The result is as follows:

```
Param_Bounds:tp_wc06_ndes.exe:exit.c:_ks@119->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_ks@121->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_ks@123->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@71->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@72->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@75->_memcpy:81:r2:r2=12
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@75->_memcpy:81:r2:r2=12
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@79->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@80->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@84->_memcpy:81:r2:r2=12
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@94->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_des@95->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_main@227->_memcpy:81:r2:r2=8
Param_Bounds:tp_wc06_ndes.exe:exit.c:_main@227->_memcpy:81:r2:r2=8
```

This shows that *r2* is indeed 8 or 12 at each call of *memcpy*. The H8/300 requires *int* data to be word-aligned and so we assume that all *immense* and *great* variables are also word-aligned which means that *memcpy* uses its word-by-word loop. For Round 2 we asserted the worst-case bound of 6 iterations for this loop, giving an overestimated WCET. The WCET bound could be sharpened in Round 3

TBA by analysing the execution time of *memcpy* for 8 octets and asserting this execution time separately for each such call of *memcpy* when the call can be identified with the assertion language. Identifying these calls is easy for calls from *main* or *ks*, because all *memcpy* calls from these functions copy 8 octets. The *des* function makes both kinds of *memcpy* calls so the 8-octet calls should be identified based on their position in loops, which is more difficult. TBA.

Analysis for the SPARC/BCC target

The compiler and library routines add no loops to those in the source program. All loops are bounded automatically in Round 1.

Options and assertions: ndes

Comp	Round	Options / assertions
gcc	1	*-lego*
	2	*-lego*

		`subprogram "_memcpy"`
		
		`-- The word-by-word loop is used.`
		`loop that executes "0E1C" repeats 6 times; end loop;`
		
		`-- The octet-by-octet loop is not used.`
		`loop that executes "0E28" repeats 0 times; end loop;`
		
		`end "_memcpy";`

iar		
bcc	1	*-leon2 -via_positive*

3.13 Program ns

Nature of the program

Searching for a given value in a 4-dimensional array by a straight-forward traversal of the array using a 4-deep nested loop. All loops are simple for-loops with static counter ranges.

Analysis problems

None. Round 1 succeeds without any assertions.

Options and assertions: ns

Comp	Round	Options / assertions
gcc	1	*-lego*
iar	1	*-lego*
bcc	1	*-leon2 -via_positive*

3.14 Program nsichneu

Nature of the program

The program simulates a Pr/T net and was generated by a code-generation tool at C-LAB, Paderborn, Germany. It contains just one function, `main`, with 126 `if`-statements, each of which introduces a local block and one local `if`-statement, for a total of 252 `if`-statements. For the H8/300 with *gcc* the `main` function contains 12 103 instructions. This would be grotesque in a hand-written program but is not unlikely for code generated from a transition-system model.

Analysis problems

The `main` function contains one loop which is a simple counted loop that Bound-T normally could handle. However, Bound-T's data-dependency analysis is not strong enough to eliminate the huge amount of other computation in the `main` function, so the arithmetic analysis bogs down and was aborted for the SPARC target. For the H8/300 and GCC target the auxiliary program (the Omega Calculator or "oc") that Bound-T uses for the arithmetic analysis detects a false assertion (possibly an internal error) and so Round 1 self-aborts for this target, but it would no doubt have been aborted for taking too long also.

Note also that the `if` statements in main have complex conditions which are translated into several conditional branches in the machine code – typically each of the 126 `if` nests generates nine conditional branch instructions.

For Round 2 we asserted the number of iterations of this (only) loop.

It is quite possible, perhaps very likely, that there are correlations between the `if` conditions, which means that there may be a large number of infeasible paths in the main function. Bound-T has at present no way to find those paths, so the computed WCET bound may be correspondingly pessimistic.

Options and assertions: nsichneu

This program is too large for the *-lego* H8/300 device so we use the largest H8/300 model, the H8/3297.

Comp	Round	Options / assertions
gcc	1	*-3297*
	2	*-3297*

		`subprogram "_main"`
		`  loop repeats 2 times; end loop;`
		`end "_main";`

iar		
bcc	1	
	2	*-leon2 -via_positive*

		`subprogram "main"`
		`  loop repeats 2 times; end loop;`
		`end "main";`

3.15 Program recursion

Nature of the program

The program computes Fibonacci numbers using a recursive function.

Analysis problems

Bound-T cannot analyse recursive programs and thus fails on this benchmark.

Options and assertions: recursion

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

Results

Bound-T reports that the program is recursive, shows the recursion cycle (function `fib` calling function `fib`) and stops without computing a WCET bound.

3.16 Program statemate

Nature of the program

This code was automatically generated by the STAtechart Real-time-Code generator STARC which was developed at C-LAB, Paderborn, Germany. The original StateChart specifies an experimental car window lift control. Thus, this program is in some ways similar to the `nsichneu` program. However, here the state-transition system is encoded as `switch/case` statements and the code is divided into 8 subprograms, including `main`.

General analysis problems

The program contains one loop, in subprogram `FH_DU`. This loop is not a counted loop; it repeats state transitions until the state is “stable”. Thus, Bound-T would not have found any bounds for the loop, and a simulation or execution of the program seems necessary for finding the number of iterations.

Analysis for the H8/300 GCC target

For this target the subprogram `FH_DU` is so complex that the arithmetic analysis (in the Omega Calculator) for Round 1 was aborted after one hour without result.

Analysis for the SPARC/BCC target

For this target the code of the subprogram *FH_DU* is not so complex and Round 1 finishes quickly, but it fails to bound the loop, as expected.

Options and assertions: statemate

Comp	Round	Options / assertions
gcc	1	*-lego*
iar		
bcc	1	*-leon2 -via_positive*

Results

As we cannot find bounds on the loop in *FH_DU* we have no WCET bound to report. We report the results for Round 0 or Round 1 just to show the number of subprograms and loops and the analysis time (for the incomplete analysis).

3.17 Program papa_ap

Nature of the program

This is the "autopilot" part of the PapaBench program for an autonomous aircraft. It executes a flight plan and communicates with the "fly-by-wire" part (program *papa_fbw*, section 3.18)which runs the servo controls.

General analysis problems

Round 1 was taking too long in arithmetic analysis and was aborted. We made a Round 0 to get the call-graph and started analysing the subprograms from the bottom up, creating assertions as necessary.

The subprogram *nav_home* has two loops (in the invocation of the *Circle* macro) and Bound-T can bound neither loop. The loops actually result from the macro *NormCourse* and are *while* loops that normalize an angle (in degrees) by adding or subtracting 360 degrees until the variable is in [0, 360). We assume and assert one repetition of each loop.

The subprogram *auto_nav* (unusually stored in the "header" file *flight_plan.h*) has eight loops that Bound-T cannot bound. Six of them arise from three invocations of the macro *Circle* and two from an invocation of the macro *Goto3D*; in the end, both macros use the macro *NormCourse*, the same as in *nav_home*, above. We assume and assert one repetition of each loop.

The subprogram *course_pid_run* has two loops that Bound-T cannot bound; they arise from an invocation of the *NORM_RAD_ANGLE* macro and are *while* loops that normalize an angle by adding or subtracting 2π. We assume and assert one repetition of each loop. The same happens in the subprogram *estimator_update_ir_estim* which uses *NORM_RAD_ANGLE* twice and thus has four such loops.

The *main* subprogram has two loops that Bound-T cannot bound: an initial loop that waits for 30 ticks of the *timer_period* function, and an eternal loop that runs the tasks. The execution time of the first loop is irrelevant (initialization) and for the eternal loop we are happy to measure one repetition, so we assert one repetition for both loops.

Analysis of the H8/300 GCC mathematical libraries

Most of the problems in the analysis for the H8/300 come from the mathematical library routines. The analysis of these routines for the H8/300 and the *gcc* compiler is described in section 3.19 below, because it is independent of the application and thus the same for the *papa_ap* and *papa_fbw* benchmarks. The major part of the rather long analysis time (see section 4) is spent on the library routine *___kernel_rem_pio2f*.

The *memcpy* function is used only from the mathematical libraries as described in section 3.19. We used the Bound-T option *-trace calls* to check that there are no other calls to *memcpy*.

Analysis for the SPARC/BCC target

We first compiled the executable with BCC using *-O2* optimization for all modules. Surprisingly, this made BCC unroll the eternal loop in the *main* function, duplicating the calls to the tasks and also making the control-flow graph irreducible. To avoid this, we reduced the optimization to *-O1* for the file *mainloop.c*, keeping *-O2* for the other source-code files. The loop in *main* is then not unrolled and more importantly the control-flow graph is reducible and thus analysable with Bound-T.

The subprogram *nav_update* consists of a call to *compute_dist2_to_home* followed by a call to *auto_nav*. The BCC compiler in-lined the call to *auto_nav* which means that assertions on loops in *auto_nav* must be written as if the loops were in *nav_update*. Moreover this is the only call to *auto_nav*, so *auto_nav* does not appear at all in the analysis as a separate subprogram and is not shown in the call-graph.

The loop in the subprogram adc_init was not automatically bounded because the loop-counter is of type *uint8_t* and the compiler masks the value with 255 after incrementing it. An assertion that constrains the value to 0 .. 255 helped Bound-T analyse the loop.

Analysis of the SPARC/BCC mathematical libraries

The mathematical libraries again posed some analysis problems. The analysis of these routines for the SPARC/BCC target is described in section 3.21 below, because it is independent of the application and thus the same for the *papa_ap* and *papa_fbw* benchmarks.

Options and assertions: papa_ap

Comp	Round	Options / assertions
gcc	1	*-3297*
	2	*-3297*

```
subprogram "_main"
   -- The initalization wait and the eternal loop.
   all 2 loops repeat 1 time; end loops;
end "_main";

subprogram "_nav_home"
   -- From NormCourse.
   all 2 loops repeat 1 time; end loops;
end "_nav_home";

subprogram "_auto_nav"
   -- From NormCourse.
   all 6 loops repeat 1 time; end loops;
end "_auto_nav";

subprogram "_course_pid_run"
   -- From NORM_RAD_ANGLE.
   all 2 loops repeat 1 time; end loops;
end "_course_pid_run";

subprogram "_estimator_update_ir_estim"
   -- From NORM_RAD_ANGLE.
   all 4 loops repeat 1 time; end loops;
end "_estimator_update_ir_estim";
```

(See section 3.19 for assertions on math library routines.)

iar

bcc

1

-leon2 -via_positive -max_par_depth 0

```
subprogram ".div"
   time 149 cycles;
   hide;
end subprogram ".div";

subprogram ".urem"
   time 154 cycles;
   hide;
end subprogram ".urem";
```

2

-leon2 -via_positive

```
subprogram "main"
   -- The initalization wait and the eternal loop.
   all 2 loops repeat 1 time; end loops;
end "main";

subprogram "nav_home"
   -- From NormCourse.
   all 2 loops repeat 1 time; end loops;
end "nav_home";

subprogram "nav_update"
   -- Inlined subprogram "auto_nav":
   -- From NormCourse.
   all 6 loops repeat 1 time; end loops;
end "nav_update";

subprogram "course_pid_run"
   -- From NORM_RAD_ANGLE.
   all 2 loops repeat 1 time; end loops;
end "course_pid_run";

subprogram "estimator_update_ir_estim"
   -- From NORM_RAD_ANGLE.
   all 4 loops repeat 1 time; end loops;
end "estimator_update_ir_estim";
```

```
subprogram "adc_init"
  -- Local variable "i" is an octet.
  variable address "p4" 0 .. 255;
end "adc_init";
```

3.18 Program papa_fbw

Nature of the program

This is the "fly by wire" part of the PapaBench program for an autonomous aircraft. It communicates with the "autopilot" part (program *papa_ap*, section 3.17) and runs servo loops to control the aircraft.

General analysis problems

Round 1 was taking too long in arithmetic analysis and was aborted. We made a Round 0 to get the call-graph and started analysing the subprograms from the bottom up, creating assertions as necessary.

The subprogram *uart_print_string* has a loop over the string parameter that is terminated by the null octet at the end of the string. The number of iterations depends on the string length in a way that Bound-T does not detect. However, this subprogram is called at only one place, with a constant string of length 65, so the loop is easy to bound with an assertion.

The *main* subprogram has an eternal loop that invokes the cyclic and sporadic tasks. We assert one execution.

Analysis problems in the H8/300 GCC mathematical libraries

The same problems occurred as for the *papa_ap* program described earlier (however, some library routines were used in one program but not in the other). We used the same assertions as for *papa_ap*. See section 3.19.

Analysis for the SPARC/BCC target

We first compiled the executable with BCC using *-O2* optimization for all modules but met the same problem with an irreducible main loop as in the *papa_ap* program. As for *papa_ap*, we reduced the optimization to *-O1* for the file *main.c*, keeping *-O2* for the other source-code files.

Bound-T could not bound the loop in the subprogram *adc_init* because the loop-counter *i* is declared as an unsigned octet variable but is held in a 32-bit register. BCC then uses a logical AND instruction to mask *i* to 8 bits after incrementing it, and so Bound-T does not detect the increment of 1. We asserted that *i* is in 0 .. 255 which lets Bound-T deduce that the AND with 255 has no effect on the value of *i* and reveals that the loop increments *i* by one, after which Bound-T finds the loop-bound. The same problem arose in the subprograms *servo_init*, *servo_transmit* and *to_autopilot_from_last_radio*, and we used the same solution for all.

Options and assertions: papa_fbw

Comp	Round	Options / assertions
gcc	1	*-3297*
	2	*-3297*

		`subprogram "_uart_print_string"`
		`   -- The loop over the string, up to the null octet.`
		`   loop repeats 65 times; end loop;`
		`end "_uart_print_string";`
		
		`subprogram "_main"`
		`   loop repeats 1 times; end loop;`
		`end "_main";`
		
		(See section 3.19 for assertions on math library routines.)

iar		
bcc	1	*-leon2 -via_positive*
	2	*-leon2 -via_positive*

		`subprogram "main"`
		`   -- The eternal loop.`
		`   loop repeats 1 times; end loop;`
		`end "main";`
		
		`subprogram "uart_print_string"`
		`   -- The loop over the string, up to the null octet.`
		`   loop repeats 65 times; end loop;`
		`end "uart_print_string";`
		
		`subprogram "adc_init"`
		`   -- Variable "i" is an unsigned octet:`
		`   variable address "p3" 0 .. 255;`
		`end "adc_init";`
		
		`subprogram "servo_init"`
		`   -- Variable "i" is an unsigned octet:`
		`   variable address "p3" 0 .. 255;`
		`end "servo_init";`
		
		`subprogram "servo_transmit"`
		`   -- Variable "servo" is an unsigned octet:`
		`   variable "servo_transmit|servo" 0 .. 255;`
		`end "servo_transmit";`
		
		`subprogram "to_autopilot_from_last_radio"`
		`   -- Variable "i" is an unsigned octet:`
		`   variable address "p4" 0 .. 255;`
		`end "to_autopilot_from_last_radio";`

3.19 Math library on H8/300 – GCC

Introduction

The *papa_ap* and *papa_fbw* programs make extensive use of floating-point computation and mathematical library routines, in particular trigonometric functions. The H8/300 has no hardware floating-point unit and so the programs

must use software routines both for the basic floating-point operations (addition, multiplication and so on) and for the trigonometric functions. The *gcc* compiler for the H8/300 uses the GNU *newlib* library (version 1.11.0). Most of the work in the analysis of *papa_ap* and *papa_fbw* was spent on analysing these library routines.

We studied the source-code of the *newlib* library to understand the structure and termination conditions of the loops. We describe the details below for the record, but they are rather repetitive. Note that many of the loops could be bounded automatically, or by much simpler or fewer assertions, if we would implement a few extensions to Bound-T, chiefly the analysis of local variable references via a frame pointer. Note also that most of the assertions for these library routines are independent of the application program, and in fact we used the same assertions for the *papa_ap* and *papa_fbw* programs. Most of the assertions should even be valid for *newlib* on other target processors with similar floating-point precision requirements.

Analysis of newlib routines on H8/300 with gcc

The routine ___*unpack_f* has one loop that Bound-T could not bound, in addition to the three which Bound-T did bound. The difficult loop seems to be some form of normalization where a 32-bit mantissa (in *r0*:*r1*) is iteratively doubled and a binary exponent correspondingly decremented until the most significant 1-bits in the mantissa are placed suitably. The logic is not easy to follow, but we assume that the loop will not iterate for more than the number of bits in a single-precision mantissa, which is 22 and which we assert as the bound.

The routine ___*pack_f* has two loops that Bound-T could not bound, in addition to the two which Bound-T did bound. Again, these loops are normalization loops, but here they are counted loops. The problem for Bound-T is that the counter limit is taken from a local variable that is addressed via register *r6* that is used as a frame pointer, something Bound-T for the H8/300 does not support at present. As for __*unpack_f* we assume and assert a limit of 22 repetitions.

The routine ___*fixunssfsi* has a loop that Bound-T could not bound. This is again a counted normalization loop, but here a bound of 30 iterations seems to be necessary.

The math library routine ___*floatsisf* has a loop that Bound-T could not bound, which is very similar to the difficult loop in ___*unpack_f* and for which we also assert 22 as the iteration bound.

The routine ___*mulsf3* has three loops that Bound-T could not bound. One is a simple counter loop (for counter in 0 .. 31) but the problem is the use of a frame pointer, as in ___*pack_f*. The other loops are shifting and adding loops, so we assume a limit of 22 iterations.

The routine __*fpadd_parts* has two loops that Bound-T could not bound, similar to the loops described above. We asserted a bound of 22 iterations. This function also calls *memcpy* to copy 8 octets, so we assert the loop-bounds in *memcpy* accordingly (assuming word-aligned data).

The routine ___*fixsfsi* has a loop that Bound-T could not bound, very similar to the counter loop in ___*mulsf3* . We assert 32 as the bound on iterations.

The routine ___*divsf3* has a loop that Bound-T could not bound. Every iteration of the loop shifts a 32-bit (possibly only a 22-bit) number one position, so we assume and assert that there are at most 32 iterations.

The routine `___kernel_rem_pio2` has 20 loops that Bound-T could not bound, in addition to the single loop for which Bound-T found bounds without assertions. Most of these loops could be bounded with some bounds on the local variables (`m`, `jx`, `jk`, `jz`) but this routine uses a frame pointer (`r6`) so we cannot assert values of local variables and must assert the repetition bounds of each loop.

The routine `_floorf` has one loop that Bound-T could not bound from a shift operation where the shift-count is bounded to 0 .. 22 by controlling conditions. However, the shifting loop probably uses only the low octet of this variable, and Bound-T does not propagate the 0 .. 22 bounds on the whole variable to the low octet.

The routine `___ieee754_rem_pio2` has two loops and Bound-T can bound neither loop. The first loop is a simple counter loop, `for (i=0; i<2; i++)`, but the local variable `i` is probably accessed via a frame pointer, which Bound-T sees as an unresolved dynamic memory access (pointer) giving an unknown value. The second loop is a `while` loop that scans an array and stops when a nonzero value is found. The array contents are unknown to Bound-T.

The routine `___ieee754_sqrtf` has two loops in the source code (*ef_sqrt.c*). Bound-T with default options (in particular *-bcc=unsigned*) considers one of these loops (the one that handles a "subnormal" value) to be unreachable. This is probably an error and so we used the safer option *-bcc=signed* for this routine. Bound-T cannot bound these loops because they have no counters. The first loop normalises a number by shifting it left until it has the bit 2^{23} set. Thus the loop repeats at most 23 times. The second loop "generates sqrt(x) bit by bit". The loop "counter" is initialized to 2^{24} and shifted right one bit on each iteration until it is zero. Thus the loop repeats 25 times. The routine also has two shift operations that create loops in the machine code; Bound-T did find bounds for these loops.

We initially analysed these routines using default Bound-T option *-bcc=unsigned* which makes some approximations in the modelling of arithmetic with signed variables that help Bound-T find loop-bounds. These approximation may be unsafe, and indeed Bound-T wrongly classified some parts of some routines as infeasible (not reachable) and gave an underestimated WCET bound. For these routines we used the safe option *-bcc=signed* (through a "property" assertion). The routines are `___kernel_rem_pio2f` and `___ieee_754_sqrt`. The safe model was also used for some other routines but there it had no effect.

Options and assertions

The library routines were analysed together with the application (*papa_ap* or *papa_fbw*) thus using the same Bound-T options as for the analysis of the application.

The following table shows the assertions that we used for the *newlib* routines that are called in the `papa_ap` and `papa_fbw` programs. Most of them identify the loops by their offset from the start of the containing subprogram, using the phrase "loop that executes +offset".

Assertions
`subprogram "___unpack_f"` `  loop that executes +"00A4" repeats 22 times; end loop;`

```
end "___unpack_f";

subprogram "___pack_f"
   loop that executes +"00B4" repeats 22 times; end loop;
   loop that executes +"00F2" repeats 22 times; end loop;
end "___pack_f";

subprogram "___floatsisf"
   loop repeats 22 times; end loop;
end "___floatsisf";

subprogram "___mulsf3"
   loop that executes +"0132" repeats 32 times; end loop;
   loop that executes +"0232" repeats 22 times; end loop;
   loop that executes +"0278" repeats 22 times; end loop;
end "___mulsf3";

subprogram "___divsf3"
   loop repeats 32 times; end loop;
end "___divsf3";

subprogram "__fpadd_parts"
   all 2 loops repeats 22 times; end loops;
end "__fpadd_parts";

subprogram "___fixsfsi"
   loop repeats 32 times; end loop;
end "___fixsfsi";

subprogram "___kernel_rem_pio2f"

   -- Fact re parameters, looking at the call of this function from
   -- __ieee754_rem_pio2f:
   --
   -- prec = 2. Consequently jk = init_jk[2] = 9 and jp = jk = 9.
   -- nx <= 3.  Consequently jx <= 2 and m = jx + jk <= 11.

   -- A condition that needs bcc=signed for correct analysis.
   property “bcc_signed” 1;

   -- for(i=0;i<=m;i++,j++)
   -- See bound on m, above.
   loop that executes +"0114" repeats 12 times; end loop;

   -- for (i=0;i<=jk;i++) (outer loop)
   -- See bound on jk, above.
   loop that executes +"01B0" repeats 10 times; end loop;

   -- for(j=0,fw=0.0;j<=jx;j++) (inner loop)
   -- See bound on jx, above.
   loop that executes +"01E4" and is in loop repeats 3 times; end loop;

   -- In the following we assume that no "recomputation" is needed. TBC.
   -- This means that jz = jk <= 9 on exit from the recomputation loop.

   -- The "recompute" loop.
   -- We assume that no recomputations are needed.
   loop that executes +"02C6" repeats 1 time; end loop;

   -- The following loops are all inside the recomputation loop so
   -- we must qualify them with "is in loop".

   -- for(i=0,j=jz,z=q[jz];j>0;i++,j--)
   -- Here jz is 9 .. 1.
   loop that executes +"031A" and is in loop repeats 9 times; end loop;

   -- In the following, we use the comment that q0 < 3 and
```

```
-- the controlling condition that q0 > 0.

-- iq[jz-1]>>(8-q0)
loop that executes +"04EC" and is in loop repeats 7 times; end loop;

-- i<<(8-q0)
loop that executes +"0516" and is in loop repeats 7 times; end loop;

-- iq[jz-1]>>(7-q0)
loop that executes +"0552" and is in loop repeats 6 times; end loop;

-- for(i=0;i<jz ;i++), where jz <= 9.
loop that executes +"061C" and is in loop repeats 8 times; end loop;

-- The following loops are on the path that "recomputes", so they
-- are never executed by our assumption of no recomputation, above.
-- However, we put bounds on them anyway, if we decide to include
-- recomputation in the future.

-- for (i=jz-1;i>=jk;i--)
-- Here i is used to index iq[20], so the range is at most 19 .. 9
-- considering the value of jk = 9.
loop that executes +"0812" and is in loop repeats 11 times; end loop;

-- In the following, we use the facts that jk = 9 and
-- jk-k is used to index iq[] so it must be >= 0, thus
-- k <= 9.

-- for(k=1;iq[jk-k]==0;k++)
-- See bound k <= 9, above.
loop that executes +"08A6" repeats 9 times; end loop;

-- for(i=jz+1;i<=jz+k;i++) (outer loop)
-- See bound k <= 9, above.
loop that executes +"091C" and is in loop repeats 9 times; end loop;

-- for(j=0,fw=0.0;j<=jx;j++) (inner loop)
-- See bound on jx, above.
loop that executes +"097C" and is in (loop that is in loop)
    repeats 3 times;
end loop;

-- Here endeth the "recompute" loop.

-- We now come to a section of code where jz can be decreased
-- by some (unknown) amount or increased by at most 1, thus
-- we will have jz <= 10 after this section.

-- while(iq[jz]==0) { jz--; q0-=8;}
-- Here jz was decremented before the loop, so jz <= 8 on
-- entry to the loop.
loop that executes +"0AD6" repeats 8 times; end loop;

-- So, as we said above, from now on jz <= 10.

-- for(i=jz;i>=0;i--)
loop that executes +"0C6C" repeats 11 times; end loop;

-- for(i=jz;i>=0;i--) (outer loop)
loop that executes +"0CFA" repeats 11 times; end loop;

-- for(fw=0.0,k=0;k<=jp&&k<=jz-i;k++) (inner loop)
-- Here k is in 0 .. min (jp, jz-i) = 9. However, the
-- upper bound on k depends on the counter i of the outer
-- loop, so this is a triangular loop.
loop that executes +"0D60" and is in loop repeats 10 times; end loop;
```

```
   -- for (i=jz;i>=0;i--)
   loop that executes +"0E30" repeats 11 times; end loop;

   -- for (i=jz;i>=0;i--)
   loop that executes +"0ECA" repeats 11 times; end loop;

   -- for (i=1;i<=jz;i++)
   loop that executes +"0F8A" repeats 10 times; end loop;

   -- for (i=jz;i>0;i--)
   loop that executes +"103A" repeats 10 times; end loop;

   -- for (i=jz;i>1;i--)
   loop that executes +"1112" repeats 9 times; end loop;

   -- for (fw=0.0,i=jz;i>=2;i--)
   loop that executes +"11F8" repeats 9 times; end loop;

end "___kernel_rem_pio2f";

subprogram "___ieee754_rem_pio2f"

   -- for(i=0;i<2;i++)
   loop that executes +"067C" repeats 2 times; end loop;

   -- while(tx[nx-1]==zero) nx--;
   -- Here nx is initialized to 3 before the loop.
   loop that executes +"071C" repeats 3 times; end loop;

end "___ieee754_rem_pio2f";

subprogram "___ieee754_sqrtf"

   -- A condition that needs bcc=signed for correct analysis.
   property “bcc_signed” 1;

   -- for(i=0;(ix&0x00800000L)==0;i++) ix<<=1;
   loop that executes +"00DC" repeats 23 times; end loop;

   -- r = 0x01000000L; while(r!=0) {...; r>>=1; }
   loop that executes +"01A4" repeats 25 times; end loop;

end "___ieee754_sqrtf";

subprogram "_floorf"
   -- 0x007fffff>>j0
   -- Use the controlling conditions j0 < 23 and j0 >= 0.
   loop repeats 22 times; end loop;
end "_floorf";

subprogram "_memcpy"
   -- Assertions for calls from fpadd_parts.

   -- The word-by-word loop is used.
   loop that executes +"0014" repeats 4 times; end loop;

   -- The octet-by-octet loop is not used.
   loop that executes +"0020" repeats 0 times; end loop;

end "_memcpy";

-- Some routines that may be sensitive to -bcc=unsigned
-- and should be analysed with the safer -bcc=signed.
subprogram "___divsf3"     property "bcc_signed" 1; end;
subprogram "___floatsisf"  property "bcc_signed" 1; end;
subprogram "___mulsf3"     property "bcc_signed" 1; end;
subprogram "___pack_f"     property "bcc_signed" 1; end;
```

3.20 Math library on H8/300 – IAR

TBA

3.21 Math library on SPARC – BCC

The analysis of the math library on this target is made easier by the assumed presence of a Floating-Point Unit which means that the basic floating-point operations are implemented by machine instructions instead of software routines.

The Gaisler Research BCC compiler that we used for this work comes with the *newlib* library version 1.13.0. Unfortunately, but following tradition, the library is provided without debugging information, which means that Bound-T cannot identify loops by source-line numbers and we cannot assert variable-values using the symbolic (C) variable identifiers. It should be possible to recompile the library with debugging information, but we did not do so for this work; instead we compared the source-code of the relevant library routines from *newlib* 1.13.0 with the source-code of version 1.11.0 as used in the H8/300 GCC compiler, and then assumed that the compiler makes minimal modifications to the loops, in particular that it does not reorder loops.

Bound-T finds 20 loops in the code of the subprogram `__kernel_rem_pio2`. The number and nesting of the code loops agrees with the number and nesting of loops in the source-code of this subprogram. The source-code of this function (in *k_rem_pio2.c*) is very similar to the source-code of the corresponding function `___kernel_rem_pio2f` in the H8/300 GCC target; the difference is mainly that the SPARC routine uses double-precision floating-point numbers where the H8/300 routine uses single precision. Moreover, the `prec` parameter has the same value (2). Most of the reasoning we used for this subprogram's loop-bounds on the H8/300 GCC target is valid on the SPARC/BCC target, assuming that the compiler does not unpeel or unroll any loops, and the same bounds on the loops result. The SPARC code has no loops for the shift operations so those assertions from the H8/300 GCC target are omitted for the SPARC.

A similar comparison between the H8/300 and SPARC versions was used to understand the loop-bounds in the subprogram `__ieee754_sqrt`. But it is a mystery why the BCC compiler does not use the SPARC FPU square-root instruction instead of this routine (perhaps another compilation option is needed).

Options and assertions

The library routines were analysed together with the application (*papa_ap* or *papa_fbw*) thus using the same Bound-T options as for the analysis of the application.

The following table shows the assertions that we used for the *newlib* routines that are called in the `papa_ap` and `papa_fbw` programs. Most of them identify the loops by their offset from the start of the containing subprogram, using the phrase “loop that executes +offset”.

Assertions

```
subprogram "__kernel_rem_pio2"

   -- Fact re parameters, looking at the call of this function from
   -- __ieee754_rem_pio2f:
   --
   -- prec = 2. Consequently jk = init_jk[2] = 9 and jp = jk = 9.
   -- nx <= 3.  Consequently jx <= 2 and m = jx + jk <= 11.

   -- for(i=0;i<=m;i++,j++)
   -- See bound on m, above.
   loop that executes +"0088" repeats 12 times; end loop;

   -- for (i=0;i<=jk;i++) (outer loop)
   -- See bound on jk, above.
   loop that executes +"00CC" repeats 10 times; end loop;

   -- for(j=0,fw=0.0;j<=jx;j++) (inner loop)
   -- See bound on jx, above.
   loop that executes +"00E4" and is in loop repeats 3 times; end loop;

   -- In the following we assume that no "recomputation" is needed. TBC.
   -- This means that jz = jk <= 9 on exit from the recomputation loop.

   -- The "recompute" loop.
   -- We assume that no recomputations are needed.
   loop that executes +"0128" repeats 1 time; end loop;

   -- The following loops are all inside the recomputation loop so
   -- we must qualify them with "is in loop".

   -- for(i=0,j=jz,z=q[jz];j>0;i++,j--)
   -- Here jz is 9 .. 1.
   loop that executes +"0158" and is in loop repeats 9 times; end loop;

   -- for(i=0;i<jz ;i++), where jz <= 9.
   loop that executes +"027C" and is in loop repeats 8 times; end loop;

   -- The following loops are on the path that "recomputes", so they
   -- are never executed by our assumption of no recomputation, above.
   -- However, we put bounds on them anyway, if we decide to allow
   -- recomputation in the future.

   -- for (i=jz-1;i>=jk;i--)
   -- Here i is used to index iq[20], so the range is at most 19 .. 9
   -- considering the value of jk = 9.
   loop that executes +"0320" and is in loop repeats 11 times; end loop;

   -- In the following, we use the facts that jk = 9 and
   -- jk-k is used to index iq[] so it must be >= 0, thus
   -- k <= 9.

   -- for(k=1;iq[jk-k]==0;k++)
   -- See bound k <= 9, above.
   loop that executes +"036C" repeats 9 times; end loop;

   -- for(i=jz+1;i<=jz+k;i++) (outer loop)
   -- See bound k <= 9, above.
   loop that executes +"03C8" and is in loop repeats 9 times; end loop;

   -- for(j=0,fw=0.0;j<=jx;j++) (inner loop)
   -- See bound on jx, above.
   loop that executes +"03EC" and is in (loop that is in loop)
       repeats 3 times;
   end loop;

   -- Here endeth the "recompute" loop.
```

```
   -- We now come to a section of code where jz can be decreased
   -- by some (unknown) amount or increased by at most 1, thus
   -- we will have jz <= 10 after this section.

   -- while(iq[jz]==0) { jz--; q0-=24;}
   -- Here jz was decremented before the loop, so jz <= 8 on
   -- entry to the loop.
   loop that executes +"04D4" repeats 8 times; end loop;

   -- So, as we said above, from now on jz <= 10.

   -- for(i=jz;i>=0;i--)
   loop that executes +"0524" repeats 11 times; end loop;

   -- for(i=jz;i>=0;i--) (outer loop)
   loop that executes +"0560" repeats 11 times; end loop;

   -- for(fw=0.0,k=0;k<=jp&&k<=jz-i;k++) (inner loop)
   -- Here k is in 0 .. min (jp, jz-i) = 9. However, the
   -- upper bound on k depends on the counter i of the outer
   -- loop, so this is a triangular loop.
   loop that executes +"058C" and is in loop repeats 10 times; end loop;

   -- for (i=jz;i>=0;i--)
   loop that executes +"0600" repeats 11 times; end loop;

   -- for (i=jz;i>=0;i--)
   loop that executes +"0648" repeats 11 times; end loop;

   -- for (i=1;i<=jz;i++)
   loop that executes +"0698" repeats 10 times; end loop;

   -- for (i=jz;i>0;i--)
   loop that executes +"06D8" repeats 10 times; end loop;

   -- for (i=jz;i>1;i--)
   loop that executes +"0720" repeats 9 times; end loop;

   -- for (fw=0.0,i=jz;i>=2;i--)
   loop that executes +"0810" repeats 9 times; end loop;

end "__kernel_rem_pio2";

subprogram "__ieee754_rem_pio2"

   -- while(tx[nx-1]==zero) nx--;
   -- Here nx is initialized to 3 before the loop.
   loop that executes +"02A0" repeats 3 times; end loop;

end "__ieee754_rem_pio2";

subprogram "__ieee754_sqrt"

   -- while (ix0)==0) ...
   -- The loop shifts ix1 left by 21 bits. This can be
   -- done at most twice before the 32-bit variable ix1 is
   -- zero, which would prevent termination of the loop.
   loop that executes +"00B4" repeats 0 .. 2 times; end loop;

   -- for(i=0;(ix0&0x00100000)==0;i++) ix0<<=1;
   -- The are 20 bits to the right of the '1' in the mask.
   -- So at most 20 left-shifts of "ix0" before this bit is
   -- hit and terminates the loop.
   loop that executes +"0124" repeats 0 .. 20 times; end loop;

   -- r = 0x00200000L; while(r!=0) {...; r>>=1; }
```

```
   -- There are 21 bits to the right of the '1' in the initial "r".
   -- So 22 right-shifts make "r" zero.
   loop that executes +"01A0" repeats 22 times; end loop;

   -- r = sign = 0x80000000; while(r!=0) {...; r>>=1; }
   -- There are 31 bits to the right of the '1' in the initial "r".
   -- So 32 right-shifts make "r" zero.
   loop that executes +"017C" repeats 32 times; end loop;

end "__ieee754_sqrt";

subprogram ".div"
   time 149 cycles;
   hide;
end subprogram ".div";

subprogram ".urem"
   time 154 cycles;
   hide;
end subprogram ".urem";
```

4 SUMMARY OF THE RESULTS

4.1 Full results

This section tabulates the results of all the analysis rounds (except the preliminary Round 0) of all benchmark programs. In the table below the columns have the following meanings:

- *Program*: The name of the benchmark program.
- *Stmts*: The number of statements in the source code, computed as the number of source-code lines that contain a semicolon (';').
- *Comp*: The name of the compiler. The remaining columns in the row report results from the executable generated with this compiler. If this column is blank, the two following columns (*Subs* and *Loops*) in the row report properties of the source code and the remaining columns are unused. The compiler also identifies the target processor as follows:

  gcc The GNU compiler for the H8/300.
  iar The IAR compiler for the H8/300.
  bcc The Gaisler Research Bare C Compiler (based on GCC) for the SPARC.

- *Subs*: The number of subprograms (functions, procedures) in the program, including the `main` subprogram. For the SPARC target the register-window trap handlers are omitted.
- *Loops*: The number of loops in the program. The number of loops in the executable is often larger than in the source code because the compiler may generate loops (*eg*. for C expressions like `n << k`) and there may be loops in library routines. For the SPARC/BCC target the loops in the irreducible library routines are not included.
- *Round*: The number of the Round. The remaining columns in this row report the results from this Round. Blank for the source-code row.
- *Bound*: The number of loops for which Bound-T found an iteration bound. Note that some loops may be counted twice, if Bound-T finds both context-independent and context-dependent bounds for the same loop.
- *Ass*: The number of loops for which we asserted iteration bounds, perhaps for each Round as in the preceding column, or for which we gave other assertions (such as variable-value bounds) that let Bound-T find iteration bounds. Blank for the source-code row.
- *AT*: Analysis time, in seconds of real (wall-clock) time, on the computer described in section 1.2. The time is measured with one used logged in but no other heavy activity. From run to run the time varies about 5-10%.
- *WCET**: This is the WCET bound that Bound-T reports, if any, in cycles.
- *Note*: Refers to the list of notes after the table.

We again remind the reader not to use the WCET bounds to draw any conclusions regarding the efficiency of the compilers, for reasons explained in section 1.

Table 1: Benchmark analysis results

Program	Stmts	Comp	Subs	Loops	Round	Bound	Ass	AT	WCET*	Note
adpcm	344		17	20						
		gcc	22	31	*1*	26		18.4		
					2	25	5	9.5	2 002 692	
		iar								
		bcc	19	18	*1*	15		2.3		
					2	15	3	2.3	803 089	
cnt	57		6	4						
		gcc	10	5	*1*	5		0.5	45 806	
		iar								
		bcc	7	4	*1*	4		0.4	19 628	
compress	184		9	8						
		gcc	16	15	*1*	2				1
					2	4	11	2.6	586 093	
		iar								
		bcc	11	8	*1*					1, 9
					2	4	4	31.5	122 249	
cover	208		4	3						
		gcc	4	3	*1*	3		6.6	7 742	
		iar								
		bcc	4	3	*1*	3	0	1.0	3 173	
crc	36		3	3						
		gcc	3	3	*1*	3		1.3	164 118	
					3	2		0.7	85 510	2
		iar								
		bcc	3	3	*1*	2		1.1		
					2	2	1	0.5	57 663	
					3	1	1	0.2	30 000	2
duff	17		3	2						
		gcc	3	2	*1*	1		0.3		3
		iar								
		bcc	3	2	*1*	1		0.1		3
edn	127		9	13						4
		gcc	12	23	*1*	11		409.3		
					2	19	4		1 514 112	
		iar								
		bcc	11	15	*1*	1		1.3		6
					2	12	3	1.3	426 779	
insertsort	19		1	2						

Program	Stmts	Comp	Subs	Loops	Round	Bound	Ass	AT	WCET*	Note
		gcc	1	2	*1*	1		0.3		
					2	1	1	0.3	7 760	
					3	1	1	0.3	3 440	
		iar			*1*					
					2					
					3					
		bcc	1	2	*1*	1		0.2		
					2	1	1	0.2	2 078	
					3	1	1	0.2	1 133	
janne_complex	12		2	2						
		gcc	2	2	*1*	0		0.4		5
		iar								5
		bcc	2	2	*1*	0		0.8		5
matmult	42		6	5						
		gcc	10	6	*1*	6		0.8	1 506 520	
		iar								
		bcc	8	5	*1*	5		0.4	615 345	
ndes	107		5	12						
		gcc	8	21	*1*	19		41.0		
					2	19	2	39.0	823 416	
		iar			*1*					
					2					
		bcc	5	12	*1*	12		1.3	82 676	
ns	15		2	4						
		gcc	2	4	*1*	4		38.1	20 976	
		iar	3	4	*1*	4		0.3	256 960	
		bcc	2	4	*1*	4		3.0	7 097	
nsichneu	1471		1	1						
		gcc	1	1	*1*	0		208.0		6
					2	0	1	215.6	104 522	
		iar								
		bcc	1	1	*1*	0				1
					2	0	1	6.8	20 756	
recursion	11		2	0						
		gcc						<0.1		10
		iar								10
		bcc	2	0	*1*			<0.1		10
statemate	630		8	1						
		gcc	8	1	*0*	0		3.6		11

Program	Stmts	Comp	Subs	Loops	Round	Bound	Ass	AT	WCET*	Note
		iar								
		bcc	8	1	*1*	0		2.5		
papa_ap	1442		47							8
		gcc	93	85	*1*					1
					2	23	54	1 224.5	20 266 762	7
					3					
		iar			*1*					
					2					
					3					
		bcc	66	45	*1*					1
					2	2	43	12.0	62 753	7
					3					
papa_fbw	429		20							8
		gcc	29	25	*1*					1
					2	9	15	6.5	1 150 867	7
					3					
		iar			*1*					
					2					
					3					
		bcc	20	6	*1*	0		1.1		
					2	3	1	1.0	9 008	7, 12
					3					

Notes:

1. Analysis taking too long; aborted.
2. Round 3 computes an underestimated WCET bound that omits all initialisation of the CRC look-up table. The reported WCET bound is the average of this value and the WCET bound from Round 1 and thus includes one initialisation, which is the right number. The number of bounded loops is one less than for Round 2 because the initialization loop is now infeasible and is not analysed.
3. The program contains an irreducible flow-graph.
4. Loop-counters originally of type `long int` were changed to type `int`.
5. The loop termination logic is too complex for Bound-T and too complex for manual reasoning. The program should be executed or simulated.
6. Analysis aborted by assertion-error message from Omega Calculator.
7. The WCET* is given for the main function, but the important WCET bounds are those for each task, listed below in section 4.2.

8. It is difficult to find the number of loops in the source-code because C macros are used extensively to create syntactical structures including loops. The preprocessed C source should be examined.
9. Arithmetic analysis with default options led to an assertion-error message (overflow) in the Omega Calculator.
10. The program is recursive.
11. Analysis in Round 1 taking too long and aborted. Results for Round 0 are included to show the number of subprogram and loops and the analysis time (for the incomplete analysis).
12. The loops that are counted as "Bounded" were not bounded automatically but by the help of assertions on variable values (stating that variables of type `uin8_t` have values in the range 0 .. 255).

4.2 PapaBench task WCETs

The following table shows the WCET bounds for the individual tasks in the PapaBench programs *papa_ap* and *papa_fbw*.

Table 2: PapaBench task WCET bounds

Benchmark	Task	WCET*		
		H8/300 gcc	H8/300 IAR	SPARC BCC
papa_ap	altitude_control_task	60 493		158
	climb_control_task	227 871		577
	navigation_task	18 944 563		46 713
	radio_control_task	330 926		3 152
	receive_gps_data_task	405 942		4 612
	reporting_task	7 876		6 444
	stabilisation_task	248 146		762
papa_fbw	check_failsafe_task	248 312		809
	check_mega128_values_task	248 367		841
	send_data_to_autopilot_task	106 364		451
	servo_transmit	2 427		1 049
	test_ppm_task	536 395		1 958

4.3 Summary results

The table below describes very briefly the overall result for each benchmark program and each target processor/compiler. The programs are ordered according to the result: successful automatic analysis (*Auto*); automatic analysis works but was improved by assertions (*Imp*); analysis requires assertions (*Ass*); failure (*Fail*).

Table 3: Summary results

Benchmark	Overall result			Comments
	H8/300 GCC	H8/300 IAR	SPARC BCC	
cnt	Auto		Auto	
cover	Auto		Auto	
matmult	Auto		Auto	
ns	Auto	Auto	Auto	
crc	Imp		Imp	Non-rectangular loop nest.
ndes	Ass		Auto	
adpcm	Ass		Ass	Most loops automatically analysed.
compress	Ass		Ass	
edn	Ass		Ass	Most loops automatically analysed.
insertsort	Ass		Ass	
nsichneu	Ass		Ass	
papa_ap	Ass		Ass	Application simple; math library complex.
papa_fbw	Ass		Ass	
duff	Fail	Fail	Fail	Irreducible control-flow graph.
janne_complex	Fail	Fail	Fail	Loop-logic too complex; loops are not counted.
recursion	Fail	Fail	Fail	Recursive program.
statemate	Fail	Fail	fail	Loop-logic too complex; loops are not counted.

5 CONCLUSIONS FOR BOUND-T

In this section we try to understand the results in terms of the current features and failings of Bound-T and to list the most important ways in which Bound-T could be improved to analyse these benchmarks better.

TBA prose text, the following is rough memos.

Should include assertions on congruence of variable (eg. even/odd, mod n = k) (program edn, memcpy loops).

Should let assertions identify loops (and calls) by the ordinal number of the loop (first, second, third, ...) in code-address order. This would make assertions more portable and stable.

Should extend loop analysis to use proxy counters, thus allowing analysis of loops terminated by pointer equality even if the absolute pointer values are not known (program edn, memcpy loops).

Should use bounds on loop counter to bound expressions involving the loop counter, within the loop and after the loop (program adpcm, loop that calls my_cos).

Should use bounds of outer loop to place bounds on induction variables for inner loops and after the outer loop.

Should deduce bounds on variables from their use as array indices (program insertsort, inner loop).

Should improve arithmetic model to handle signed variables better (program edn, latsynth loop, also the problems with option -bcc=unsigned for the H8/300).

When an assertion or analysis bounds the value of a multi-octet variable to a value that fits in a smaller number of octets, should automatically derive bounds on these "shorter" parts of the variable and on the "extra" octets (program edn, ">>" loops, where the compiler uses only the low octet of the 16-bit variable that defines the shift count).

Should support the use of frame pointers in gcc for the H8/300.

Should support simple forms of compiler/linker name "mangling", for example the underscore that some versions of *gcc* add to an identifier, changing `main` to `_main`. This would make assertions and command-lines more portable between targets.

Should perhaps try to find iteration bounds on loops that shift a value left or right until it satisfies a certain condition (eg. is zero). Such loops are rather frequent in the floating-point libraries and occasionally occur in other parts of embedded programs.