WORST-CASE EXECUTION TIME ANALYSIS FOR DIGITAL SIGNAL PROCESSORS

Niklas Holsti, Thomas Långbacka and Sami Saarinen

Space Systems Finland Ltd.
Kappelitie 6, FIN-02200 ESPOO, FINLAND
Tel: +358 9 61328600; Fax: +358 9 61328699
e-mail: {Niklas.Holsti,Thomas.Langbacka,Sami.Saarinen}@ssf.fi

ABSTRACT

We present ongoing work to develop a software tool for estimating worst-case execution times for real-time, embedded programs. The tool applies static analysis to executable machine-code programs. Currently we mainly aim at supporting the TSC-21020E Digital Signal Processor, but the tool is designed to be easy to adapt to other target processors as well.
Contrary to most other WCET tools we attempt (whenever possible) automatic estimation of loop bounds. We also provide a rich assertion language, which can be used to set bounds on loops that the tool itself cannot bound.

1 INTRODUCTION

Space Systems Finland (SSF) is developing a software tool called BOUND-T for estimating worst-case execution times (WCET) for real-time, embedded programs. The tool applies static analysis to executable machine-code programs.

Real-time systems set hard deadlines for the software, specifying how quickly a response to some external stimulus has to be provided. Failing to meet such deadlines may have catastrophic consequences. The real-time performance of a system depends on several factors, e.g. the scheduling of concurrent tasks with different priorities. One major factor is the WCET of individual tasks. The WCET of a task may be estimated through testing, but this option is often not available at early stages of development, since testing often requires complex testing equipment and system simulators. This is why WCET tools based on static program analysis, such as ours, are attractive.

Currently the work is mainly directed towards the TSC-21020E Digital Signal Processor under contract to the European Space Agency (ESA). The TSC-21020E is a space-oriented implementation of the Analog Devices ADSP-21020 architecture. However, the tool has been designed in a modular fashion, separating processor-dependent and -independent parts. Our aim is to offer a commercial tool supporting several different target architectures and host platforms.

Contrary to most other WCET tools presented in the literature we attempt automatic estimation of loop bounds (i.e. upper bounds for the number of times the body of a loop is executed). In general, this is an unsolvable problem (a special case of the halting problem) but for typical loops a bound can often be found by data-flow and data-range analysis.

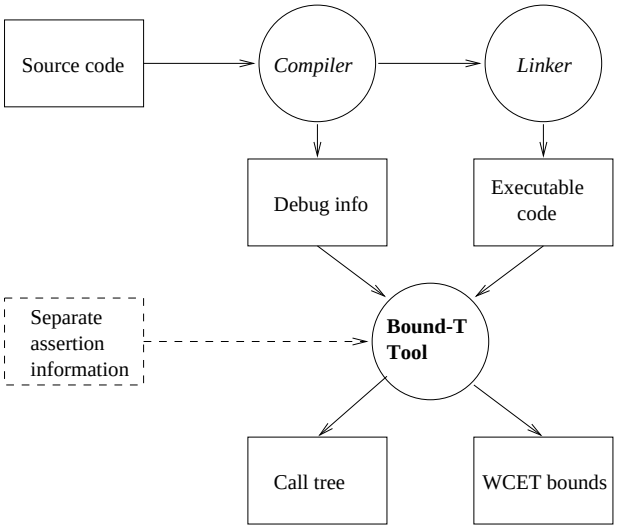

Figure 1: Context of the BOUND-T tool

2 OVERVIEW OF THE TOOL

This section presents the main design decisions that define the BOUND-T tool, highlighting the characteristics of DSP architectures and applications.

2.1 Object Code or Source Code

The task of computing the WCET for a program (or subprogram) in general means analysing the execution time of all the machine instructions on the worst-case path through the program. Thus, a simple (and general) solution is to perform the WCET calculation based directly on the object code read from an executable program file (e.g. a COFF file). This is the approach chosen in BOUND-T (see figure 1 which describes the context in which BOUND-T is used). Other researchers (see e.g. [4]) have chosen to perform parts of the analysis on the source code level (e.g. worst-case path analysis) and parts on the object level. There are also approaches

Niklas Holsti, Thomas Långbacka and Sami Saarinen. WORST-CASE EXECUTION TIME ANALYSIS FOR DIGITAL SIGNAL PROCESSORS. *Preprint of article that will appear in the proceedings of EUSIPCO'2000.*

```
#define VECTOR_LENGTH 100

float sum_vector (float vector[])
{ int i;
  float sum;
  sum = 0.0;
  for (i = 0; i < VECTOR_LENGTH; i++)
    sum += vector[i];
  return sum;
}
```

Figure 2: Simple C-function

where the analysis is performed completely on the source code level (see e.g. [3]). As BOUND-T uses object code, it is independent of the programming language. DSP applications are often written partly in assembly language but can still be analysed by BOUND-T.

Consider the C-function in Figure 2. Assuming that the function is part of an executable program called `summer.exe`, we can obtain the WCET of the function by executing BOUND-T with the command

```
boundt summer.exe sum_vector
```

The following sections explain how BOUND-T analyses the program.

2.2 Control-Flow and Call-Graph Analysis

The first analysis step is to construct the subprogram's control-flow graph by starting from the entry point and locating all branch instructions and return instructions. For a DSP this step is not so simple as it sounds, because DSPs often have advanced, complex control-flow instructions such as zero-overhead loops and architecturally-visible instruction pipelines.

In the 21020, for example, the instruction "DO address UNTIL condition" sets the processor in a state where fetching the instruction at the given address makes the processor decide whether to repeat or terminate the loop at this address. In other words, from this address control may either continue onwards, or loop back, without an explicit branch instruction, depending on the value of the given condition flag two instruction cycles earlier. Such loops can be nested and can interact in interesting ways with the delayed branch instructions. To construct the control-flow graph the tool must model the state of the processor's program sequencer, not just keep track of the current program counter. For the 21020 we model the three-stage fetch-decode-execute pipeline and the loop stack.

During control-flow analysis, call instructions are detected and the call-graph (see figure 1) is built. WCET analysis is done in a bottom-up order in the call-graph (recursion is forbidden).

2.3 Loop Bounding

Bounding the number of times the body of a loop can (at most) be executed is the major source of complication in automated WCET analysis. Typically, the user of a WCET tool is required to provide the upper bounds to the tool. In BOUND-T the aim is to minimise the effort of the user, and instead find upper bounds automatically through data-flow analysis.

The loop bodies are analysed to identify loop counters and termination conditions. The effect of an instruction is modelled as a function over the program state (much in the same way as in state-based formalisms for program verification). The state is a tuple containing all the cells (e.g. registers, processor status flags) that are affected by the instructions in the loop body. The combined effect of a sequence of instructions is computed by function composition. Control-flow branching is modelled with function domain restriction and control-flow joins with set union.

In BOUND-T a system called Omega[1] is used to express these functions. Omega implements Presburger Arithmetic, a decidable subset of integer arithmetic.

Once the loop body's combined effect has been expressed by means of a number of function compositions, it is possible to construct functions that check, for example, whether a certain variable is incremented by a constant value each time the loop body is executed, and what that constant increment is. These methods are incomplete, of course, but they work for many looping constructs that are common in DSP applications.

The automatic loop bounding is restricted to what we call *counter-based* loops. A counter-based loop is a loop that always increments (or decrements) a counter value on each iteration (and where the increment or decrement is a constant value), and terminates when the counter becomes greater than (or less than) a constant limit (assuming that the limit and original value of the counter can be found). The for-loop in the program in figure 2 of course fits this description. However, syntactically a counter-based loop does not have to be a for-loop, the important thing is the form of the loop's exit-condition and that the loop counter is updated with a constant value.

Sometimes the (actual) parameter values provided to a subprogram determine the bound of some loops in the subprogram, and a general WCET for the subprogram cannot be derived. For such subprograms, BOUND-T analyses each call of the subprogram, tries to bound the actual parameter values in the call, and estimates the WCET for this particular call, by reanalysing the subprogram in this context. This is the only form of inter-procedural data-flow analysis in BOUND-T.

2.4 User-Provided Loop Bounds

Most WCET tools presented in the literature depend on the user to provide upper bounds for loop execution. This is typically done in two different ways:

[1] Pugh, William et al. The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs. University of Maryland. Available from the URL: `http://www.cs.umd.edu/projects/omega`

```
#define VECTOR_LENGTH 100

int bin_search (int *vector, int val)
{ int low, high, mid;
  low = 0;
  high = VECTOR_LENGTH - 1;
  while (low <= high)
  {
    mid = (low + high) / 2;
    if (vector[mid] == val)
      return mid;
    else if (vector[mid] < val)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}
```

Figure 3: Complex C-function

- Through assertions in the source code of the program. In [4] a very expressive "meta-language" for providing bounds via the source code is presented.

- By providing the information interactively in response to the tool as it finds loops that need to be bounded. This introduces the problem of identifying the loops to the user. Especially if the analysis is done directly on object code, there might be no immediate way of mapping e.g. compiler-optimised code to source statements.

When BOUND-T cannot bound a loop automatically, it emits a message that explains the context of the loop: in which subprogram the loop is found, whether it is nested within some other loop, and how many loops were found before it in the same subprogram. Of course, the source-code line-numbers are also shown, if the compiler has provided this information in the executable file (the debug information shown in figure 1).

The user must provide *assertions* (see figure 1) that advise BOUND-T about the bounds for these loops. The assertions are written in a precise notation in a text file. Embedding assertions in the program source-code is not currently supported, for two reasons. Firstly, the ADI compiler for the 21020 does not generate line-number maps for optimised code, so there is no easy way to map source-embedded assertions to object-code loops. Secondly, it is often useful to compute the WCET for different scenarios, for example different interrupt loads, and these scenarios can now be defined and modified in assertion files without spurious changes to the source-code files.

The assertion notation was made as generic as possible to support programs written in any source-language, including assembler. The notation identifies loops using the logical structure of the program. This makes it meaningful to the programmer and insensitive to small changes in the code. The notation is also designed to be robust with respect to optimisation, by using loop characteristics that are likely to be left untouched in the optimisation process.

As an example consider the function in figure 3. It contains a loop that is clearly not a counter-based loop (as defined in section 2.3) since there is no counter that is incremented or decremented by a constant value, nor is there a constant limit to test against in the exit condition. Assuming that the binary search function is defined in a file called `binary.c` the following assertion states that for any call of the function `bin_search` the loop executes at most 7 times.

```
file "binary.c"
subprogram "bin_search"
loop repeats <=7 times; end_loop
```

Since the loop in `bin_search` is the only loop of the subprogram, the single keyword `loop` (as opposed to the keyword `loop_that` followed by further characteristics of the loop) is enough to identify this loop.

Assuming that the assertions above are placed in a file called `prog.a` and that the binary search function is compiled and linked into a binary file called `prog.exe` the following command gives us the WCET of the `binary_search` function.

```
boundt -assert prog.a prog.exe bin_search
```

More generally, one can specify the following kinds of assertions:

- Variable value range (minimum, maximum or both) or invariance of the variable.

- Worst-case execution counts for a statement (in practice this applies to loop bodies) or calls to a particular subprogram.

- Worst case execution time (of a subprogram or a particular call of a subprogram).

The assertions are stated for a specific *scope* which is either a subprogram, a loop or a call (or possibly the whole program in the case of variable range assertions). A particular subprogram is easy to characterise via its name. For loops and calls it is more difficult. To characterise loops the assertion syntax allows one to express such characteristics as:

- that the loop is contained (nested) in another loop

- that the loop contains another loop

- that the loop calls a particular subprogram

Characteristics can also be negated, if one wants to single out a loop that e.g. does *not* call a particular subprogram. Particular calls of a given subprogram can be characterised by stating e.g. that the call is within a loop that is again characterised as above.

Without giving any more detail about the syntax the example in figure 4 identifies a loop that has the following properties: it is in a loop that calls `"Foo"`, it contains a loop that itself does not call `"Bar"` but does call `"Fee"`, and it does not contain a loop that calls `"Fee2"`. It then gives an upper bound on the number of times the loop can repeat once it's been entered.

```
loop_that
   is_in (loop_that calls "Foo")
   and contains (loop_that not calls "Bar"
                          and calls "Fee")
   and not contains (loop_that calls "Fee2")
repeats 10 times end_loop
```

Figure 4: An example of a fairly complex loop characterisation as a part of a loop bound assertion

2.5 Worst-Case Timing Analysis

When control-flow and loop bounds have been analysed, it remains to find the worst-case execution path and its execution time. This is much simpler if the execution time of a given instruction is independent of the preceding execution path, but that is untrue for many modern processors with large caches, branch prediction or other dynamic, history-dependent optimisations. Fortunately, DSPs and embedded processors in general tend to be simple in this respect, especially in the space domain, and thus BOUND-T currently assumes constant instruction execution times. For a dynamic architecture, the tool's results are overestimates but still upper bounds on the WCET.

Most tools presented in the literature (see e.g. [1, 2, 4, 5]) calculate the WCET times using Integer Linear Programming (ILP). It is fairly straightforward to encode the control-flow graph and loop bounds into an ILP problem as shown in e.g. [1]. The tool lp_solve[2] is a popular choice as ILP solver, and BOUND-T uses it to solve the ILP problem automatically.

3 CONCLUSIONS AND FUTURE WORK

We have described BOUND-T, a tool for estimating WCET bounds of programs for the TSC-21020E DSP. The tool has been implemented but is not yet complete. To verify that the tool design is adaptable to other architectures, we also implemented a version for the Intel-8051 processor family, an 8-bit architecture common in small embedded systems. BOUND-T is implemented in the Ada language and currently runs on Sun Solaris platforms, but will be ported to PC platforms.

3.1 Implementation Status

The current version of the tool implements the functionality described above except for automatic loop bounding. Although automatic loop bounding is the most complex function planned for the tool, we are confident that it is feasible.

We have designed the loop-bounding algorithms and the necessary data-flow analysis for counter-based loops (see section 2.3). Manual tests of the algorithms and experiments with the Omega tool show that the information needed to bound counter-based loops can readily be extracted. The implementation is estimated to be ready by October 2000.

[2] By Michel Berkelaar. Available from the ftp server of the Information and Communication Systems Group, Electrical Engineering Department, Eindhoven University of Technology, from the URL: `ftp://ftp.ics.ele.tue.nl/pub/lp_solve`

The computation time of the analysis itself may be a concern, since the time complexity of the algorithms for Presburger arithmetic is known to be high. The expected analysis time on the order of a few minutes of workstation time seems acceptable, since WCET analysis will probably not be used in a rapid edit-compile-analyse cycle.

3.2 Cache Modelling

Much of the recent work on WCET tools presented in the literature focuses on modelling cache-memories and accurately predicting the execution times of individual instructions in case of cache misses etc. (see e.g. [2, 5]). This important aspect has not yet been given much attention in BOUND-T. Specifically for the TSC-21020E, we have not considered accurate cache modelling to be important (due to the limited cache size), but in order to give very precise (i.e. not too pessimistic) WCET bounds – and especially to support processor architectures with more dynamic features – this aspect has to be given attention.

3.3 Commercialisation

We are working to make BOUND-T into a commercial product. This includes ports to other target processors and host platforms. Currently we are in the stage of starting a market analysis, to look for – among other things – suitable target processors for which to port BOUND-T.

References

[1] Yay-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 298–307. IEEE Computer Society Press, 1995.

[2] Yay-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263. IEEE Computer Society Press, 1996.

[3] Patrik Persson and Görel Hedin. Interactive execution time predictions using reference attributed grammars. In *Proceedings of the Second Workshop on Attribute Grammars and their Applications (WAGA'99)*, pages 173–183, 1999.

[4] Peter Puschner. Worst-case execution time analysis at low cost. Research Report 10/97, Institut für Technische Informatik, TU Wien, 1997.

[5] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 144–153. IEEE Computer Society Press, 1998.