pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems

Guillem Bernat, Antoine Colin, Stefan Petters
Real-Time Systems Research Group
University of York. England, UK
{bernat,acolin,petters}@cs.york.ac.uk

January, 2003

Abstract

This paper describes the tool support for a framework for performing probabilistic worst-case execution time (WCET) analysis for embedded real-time systems. The tool is based on a combination of measurement and static analysis, all in a probabilistic framework. Measurement is used to determine execution traces and static analysis to construct the worst path and effectively providing an upper bound on the worst-case execution time of a program. The paper illustrates the theoretical framework and the components of the tool together with a case study.

1 Introduction

There are two main approaches for the determination of the worst-case execution time of a real-time program. Static analysis and measurement. Static analysis relies on a timing model of the hardware and attempts to determine an upper bound on the longest path of the program. Techniques include tree-based approaches [10, 4], or path based approaches [8, 12, 13]. Efforts on WCET analysis are on determining the effect of advanced processor features like cache, branch prediction and pipelines and their interactions [9, 6, 5, 7], However, these approaches are very complex as the processors themselves become more difficult to predict. An alternative approach to static analysis is by measurement. In this approach the code is run under exhaustive test conditions and the longest execution time recorded.

Both approaches have their advantages and disadvantages. Static analysis provides a safe upper bound guaranteeing that the worst case is never underestimated. This is adequate for simple programs running on simple 8 bit CPUs, however for more complex programs which are data dependent and for advanced CPU's with acceleration features like cache, pipelines, branch prediction buffers and out of order execution the analysis is extremely difficult to perform and

results in unacceptable levels of pessimism. An additional criticism of the approach is that it is based on an abstraction of the processor and may fail to capture effects that occur in the real system. Measurement approaches do observe the real system and therefore are able to account for these phenomena, however they may fail to capture the worst case as the set of test cases that may lead to the worst case may be very difficult to determine. In addition, a safety margin is usually included in the analysis, however there is no scientific process by which such safety factors can be determined.

In addition, traditional static approaches to WCET are too focused on obtaining an absolute upper bound on the execution time of the program. This may be unnecessary pessimistic if the probability of such event happening is extremely small. In probabilistic hard real-time systems the aim is to provide estimates that the probability of missing a deadline is of the same order of magnitude that other dependability estimates. For instance, probabilities smaller than 10^{-12} of missing a deadline should be provided. For such estimates to be made, it is first required to determine the probability distribution of the execution time of individual tasks.

This paper presents the pWCET framework, a theory and its tool support for probabilistic WCET analysis of real-time embedded programs. pWCET combines the best features of both measurement and analysis and allows to draw the benefits from both approaches. The framework is based on determining the execution times of individual blocks by observing the real-system (instead of relying on a processor model) but combining the worst case effects observed locally using static analysis techniques. In this way, no timing model of the processor is needed because the timing information is determined by measurement. There have been some initial approaches for probabilistic timing analysis of systems, [3, 2, 11] use extreme value statistics to model the tail of the distributions.

A different approach is the one presented by the same authors in [1]. This paper presents the general overview of the theory but its main purpose is the description of the tool support. The paper illustrates these concepts with a case study at the end of the paper. The following section provides an introduction of the theory of probabilistic WCET analysis and the description of the tool, its components and features is deferred to section 3.

2 Probabilistic WCET analysis

The aim of probabilistic WCET analysis is to determine the probability distribution of the worst-case execution time of a particular code fragment. The problem is formulated (and solved) in terms of a syntax tree representation of the program and a probabilistic timing schema.

A syntax tree is a representation of a program. It is a tree where the leafs are basic blocks (sequences of instructions that have no control flow instructions except possibly at the end) and inner nodes that correspond to syntactic composition of blocks: Sequential composition, conditional composition and iterative composition.

A timing schema is a set of rules that allow to determine the execution time of a program segment as a function of the execution time of its components. Each rule of the timing schema is associated to a type of node in the tree. For instance a trivial timing schema for static WCET analysis is as follows:

- $W(\mathsf{A}) =$ integer if A is a basic block.
- $W(\mathsf{A;B;}) = W(A) + W(B)$. Sequence of blocks.
- $W(\mathsf{if\ E\ then\ A\ else\ B\ end\ if}) = W(E) + \max(W(A), W(B))$. Conditional.
- $W(\mathsf{for\ E\ loop\ A\ end\ loop;}) = W(E) + n(W(A) + W(E))$. Loop, where n is the maximum number of iterations of the loop.

The aim of the pWCET approach is to provide an equivalent timing schema where integers are replaced by probability distributions and operations on integers are replaced by operations on random variables.

The problem of probabilistic WCET analysis is therefore:

1. To construct a syntax tree representation of the program. For illustrative purposes we consider basic blocks as the smallest execution unit, however there is no reason why other units (larger or smaller) could be used for the purpose,
2. to determine probability distributions of the individual executions of the blocks and their dependency,
3. to determine a probabilistic algebra to manipulate probability distributions,
4. to determine which and when to combine the probability distributions of the individual building blocks to derive the probability distributions of the nodes in the tree,
5. to present and visualise the results to the user.

In the rest of the section we concentrate on items 2 and 3 about the probabilistic issues, the rest of the items are discussed in the following section.

In order to provide a probabilistic timing schema we need to define equivalent operators to the sum and max for random variables. The most important fact is what assumptions about the dependence between blocks can be made.

2.1 Sequential execution $Z = X + Y$

The statistical formulation of the problem is as follows. Let X, Y be random variables that describe the execution time of a program segment. Let $F(x) = P[X \leq x]$, $G(y) = P[Y \leq y]$ be their distribution functions. Consider that situation in which X and Y are the random variables of two code segments A and

B that are executed in sequence: A;B;. Lets denote Z the random variable that describes the execution time of the sequence. The question is to determine what is the probability distribution of Z. Clearly Z can be formulated in statistical terms as $Z = X + Y$, and therefore we are interested in computing $H(z) = P[X + Y \leq z]$.

2.1.1 Dependency

One of the major issues for pWCET analysis is the determination of the dependency between X and Y. This dependence can be:

- X and Y are (assumed to be) independent,
- the joint distribution of X and Y is known and therefore the precise dependence between X and Y is also known,
- the dependency between X and Y is not known (the general case) and it can not be assumed that they are independent.

The hypothesis of independence is commonly assumed in other probabilistic analysis frameworks, however, in the framework of probabilistic WCET analysis this hypothesis is in the general case wrong. As reported in [1] making the hypothesis of independence may lead to severe underestimations of the probability of the worst case, overestimations of various orders of magnitude are possible. This is the case, for example, when the condition that makes one block to run for the worst case is the same that forces the other one for the worst case too.

The joint distribution captures precisely the exact dependence between X and Y. However, capturing such dependence by measurement is very difficult, not only for the computational complexity but also because of the nature of the process. Determining distributions of individual blocks is a difficult task because of the rare occurrence of extreme events, observing combinations of rare events in joint distributions makes the problem much harder.

In any case, there are situations where the joint distribution is not available because it can not be computed and therefore some assumption of the worst dependence between X and Y should be made. The situation when two random variables are positively correlated is called "comonotonicity". This means that both can be expressed as a non-decreasing function of another random variable U ($X = f_1(U)$ and $Y = f_2(U)$) meaning that the values of X are large when the values of Y are large too and as a consequence the probability of the extreme is not the product of probabilities (but the minimum of them).

The determination of $H(z)$ as the cumulative distribution of $Z = X + Y$ can be therefore performed as follows:

If X and Y are independent (or the assumption that they are independent is feasible) then H can be computed by performing the standard convolution between F and G:

$$H(z) = \int_x F(x)G(z - x)dx$$

If the joint distribution between X and Y is known and given by $J(x, y) = P[X \leq x, Y \leq y]$, then the distribution H can be computed as follows:

$$H(z) = \int_{x+y=z} j(x, y)$$

where $j(x, y)$ is the joint probability density function of $J(x, y)$.

Finally, if the dependence between X and Y is not known, we assume that the random variables are comonotonic. The distribution in this case is given by:

$$H(z) = \int_{x+y=z} \frac{\partial^2 \min(F(x), G(y))}{\partial x \partial y}$$

It may be the case that even the comonotonic case is not the adequate hypothesis to make about the dependency between the random variables. In such a case a general bound on the distribution of $H(z)$ should be provided that determines a limiting distribution for Z given any possible dependency. This is one of our current lines of research and a paper describing this analysis is under preparation.

The same results can be extended for an arbitrary sequence of blocks (or random variables): $Z = X_1 + X_2 + \cdots + X_n$. For details see [1].

2.2 Conditional execution $Z = \max(X, Y)$

The above discussion has shown how the distribution of the sequence of blocks can be computed probabilistically. The other main construct in the syntax tree is the conditional execution. In that case the formulation of the problem is very similar. Let X, Y and T be random variables that correspond to the expression, true and false parts of a conditional execution of E, A and B of a program segment of the form if E then A; else B; end if;. Let Z denote the distribution of the sequence. Z can be described as $Z = E + \max(X, Y)$. Where $\max(X, Y)$ is the distribution of the maximum of two random variables. The distribution $H(z)$ is given by:

$$H(z) = \int_{\max(x,y)=z} \frac{\partial^2 \min((F(x), G(y))}{\partial x \partial y}$$

The same approach can handle other types of constructs including other types of conditionals,including case statements.

2.3 Iteration

The operation for loop constructs is a combination of conditional and sequential composition. The only requirement is the identification of the maximum number of iterations of a loop, denoted by n. Then, if X, Y are the random variables that correspond to the expression guard of a loop and the body of a loop of the form for E loop B then, the distribution of the sequence Z is given by

$$Z = E + \overbrace{(E+B) + \cdots + (E+B)}^{n}$$

Other types of loops can be analysed in a similar way. The only requirement is the ability to determine maximum number of iterations of loops. Calls to other subprograms are handled by considering the call as a basic block and using the distribution of the execution time of the subprogram.

As the distributions of individual blocks do not follow standard distributions a numeric approach is the only effective solution.

2.4 Determining probability distributions

The second issue to address is to determine the actual distributions of the execution times of individual units. We use a measurement approach in which the program to be analysed is run under a large number of tests scenarios and the execution time recorded from which the probability is determined. This is a frequentist determination of probability. Other approaches are possible, more in the line of reliability analysis, the distribution could capture the distribution of the execution time on a "per incident" basis instead on a "per run" basis. In this case, the distribution of execution time is determined under particular situations (incidents) only. For example, at the critical instant or when a mode change is requested. This makes it easier to reason on probabilities of missing a deadline on a "per incident" way rather than a "per hour" measure. The method and tool described is transparent to both types of distributions, the only implication is the interpretation of the results.

It is generally easy to determine the distribution of a particular piece of code, however joint distributions are a much harder problem. The most difficult problem is that the number of experiments to perform needed to determine the probabilities grows quadratically. Besides, there are blocks in the tree for which it is not possible to determine the joint distribution because of lack of data or because the elements are not in the same level in the tree.

3 pWCET

A theory is of little use if it can not be put into practice. We have implemented the above framework into a complete toolset for probabilistic WCET analysis that covers the whole process. From automatic code analysis and syntax tree construction, to trace generation and evaluation to an efficient probabilistic calculation engine.

The pWCET toolset has the following features:

- Portable: There is a minimal dependence on the processor architecture. The structure of the program is extracted from the object code representation which requires minimal changes to a parser for different architectures. The determination of timing information is done by trace analysis and therefore a timing model of the processor is not required.

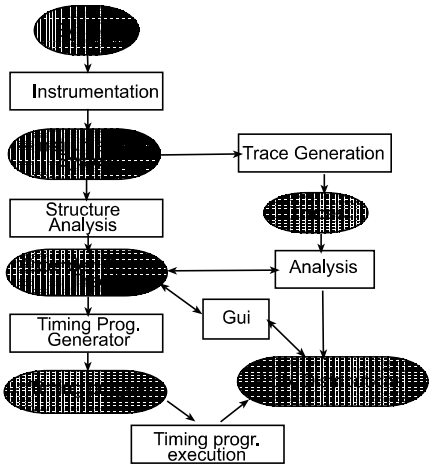

Figure 1: Overview of the pWCET toolset

- Fully flexible timing program generation: The generation process is user programmable and therefore allows different types of timing programs to be produced. The tool is able to generate both static (integer) timing programs, probabilistic programs and symbolic programs. In this paper we describe the probabilistic framework only.
- Generic: The source of the data for tracing analysis can be provided in different ways. For experimental purposes the trace can be generated using a processor simulator or by directly measuring the execution of the code on the target platform.
- Automatic loop analysis: maximum number of iterations of loops are deduced automatically from trace analysis.

The general process of the analysis and tool components is described in Figure 1.

The stages of the analysis are as follows:

- Structure analysis: the program is analysed and a syntax tree representation of the program is generated.
- Instrumentation: insertion of calls of a trace logging mechanism into the program.
- Trace generation: this step produces execution traces which capture the execution times of individual blocks for different runs of the program.
- Trace analysis: traces are parsed and distributions of individual blocks produced. Also joint distributions are captured and loop analysis determined.

- Timing program generation: a traversal of the tree generates a program that will compute the WCET of the program.
- Timing program execution: calculation of the WCET.
- Analysis of results: graphical user interface for browsing the program under analysis and the visualisation of the probability distributions.

The following subsections review each of these stages and the tool support in detail.

3.1 Instrumentation and trace generation

The aim of the instrumentation stage is to enable obtaining execution traces. An execution trace is a list of pairs (instruction,timestamp) that describe the time at which a particular instruction in the program was executed for a particular run. From this execution trace the path that the program followed as well as the different timing of the block is determined.

There are two modes of analysis, using a cycle accurate processor simulator or by directly executing the program on the target architecture. If using a cycle accurate simulator, the program does not need to be instrumented as the trace is produced by the simulator. The structural analysis determines which are the starting and ending addresses of each block of code and by parsing a log of the execution trace produced by the simulator it is able to produce the execution traces.

If the traces are determined by direct observation of the program then a mechanism to determine execution traces has to be embedded into the program and support by the OS included. This is done by manually or automatically inserting instrumentation calls into the source code, or by automatically adding the instrumentation code into the already compiled assembly code. The execution of the code results in a set of observed execution traces. The traces need to be extracted from the target hardware. These traces can then be processed by the pWCET tool (at the time of writing, the automatic program instrumentation is not yet functional).

The current demonstration version uses the simplescalar processor simulator to generate the traces, however the tool also accepts traces generated externally. The simplescalar is a MIPS cycle accurate simulator [1]. The MIPS processor has two levels of cache (second level is an integrated cache) as well as branch target buffers and out-of-order execution. Simplescalar allows the configuration of several processor configurations like changing cache size and arrangement, memory latencies, branch target prediction sizes and algorithms, etc. This is very useful to evaluate the impact of such features on the WCET of a program.

[1] http://www.simplescalar.com

3.2 Structure analysis

The structural analysis reads the non-stripped object code of the program(s) under analysis and builds a control flow graph. The program is first disassembled and the assembly code analysed. By manipulating the code at assembly level, transformations of the code included by the compiler are captured. The control flow graph is then converted into a syntax tree, called the extended syntax tree (XST). It may be the case that there are irreducible constructs (usually generated by the compiler), in this situation the analysis assumes that the whole section of the code is a block. Portability of the whole approach to a different machine architecture requires the rewriting of the lexical and syntactic analyser of assembler code which is a small task. By analysing the code at the object code level, there are no dependencies on the programming language used or only minimal.

The XST is stored as an XML file, structured as a set of trees which are made up of five types of nodes: (a) basic blocks, (b) sequences, (c) conditional code (d)loops and (e) calls. A tree is build for each subprogram, and therefore the XST of a program is made up of a series of such trees, the first one being the main program. The structure analysis also adds into the XST information for each node regarding which sections in the code it corresponds to, as well as annotations present in the source code.

3.3 Trace analysis

The trace analysis computes the distribution functions of each node in the tree from the execution traces. It uses information in the XST to determine the set of addresses that mark the start and end of each block and parses each trace accumulating the result over multiple traces. The result of the analysis is a set of execution time profiles (or ETP for short) which correspond to the discrete probability density function of individual blocks.

For selected pairs of nodes, the trace analysis is also able to determine the joint distribution function of pairs of nodes. The list of pairs of nodes to analyse is indicated before the analysis starts in a configuration file. The result of the analysis is a set of joint execution time profiles (or JEP for sort) which correspond to the discrete joint probability density function of pairs of blocks.

This is a computationally very expensive process. There may be available large number of execution traces, each one holding information of potentially long executions of the program. For example, if an execution trace records in average one every 10 instructions, then a program that runs for 1 second on a 10 MHz machine may generate up to 10^6 sampling points per second. Tests involving several hours of computation should be expected. In order to address the computational complexity the process of trace generation and analysis has been parallelised and the current implementation is able to generate the traces in a local mode (single node) or on a distributed mode using a Beowulf cluster. A special program running on a node of the cluster is responsible to distribute the work to the different nodes and merge the results after the computation has

been performed.

A second component of the trace analysis is loop identification. The information of which blocks form a loop and the nested loop structure is extracted from the syntax tree. From this information, a loop trace can also be generated. A loop trace is an indication of the index counters of each loop hit for a particular run of the program. From this loop trace, the maximum loop iteration for each loop is extracted.

3.4 Timing program generator

pWCET has a powerful mechanism for computing the WCET of programs. This is based on separating the timing analysis into a program generation part and an execution part. This enables different types of analysis to be implemented using the same framework by providing different timing program generators.

The timing program generator traverses the tree in postorder and applies the timing schema rules to each node in the tree. The result of such procedure is a set of commands on how to compute the timing program for the given tree.

The user can direct the way the analysis is performed and which rules are applied by directly manipulating the tree and modifying the rule associated to each node. For example, one common assumption is to rewrite non-rectangular loops to indicate precisely the exact number of iterations of a block, not the (possibly) pessimistic estimate obtained by the loop analysis.

We have experimented with different formats. We currently are able to generate timing programs as Ada programs, matlab scripts and ml programs.

Matlab scripts are very helpful because it allows for fast prototyping and experimentation with different operators, however in general the computation is very slow compared to a custom build solution.

The timing program reads the distributions of its sons and computes the distribution for each node in the tree. We have implemented all probabilistic algebra very efficiently exploiting the properties of the sparse data structure used to capture probability distributions. As an illustration, a convolution of a discrete distribution in Matlab can take as long as 10 times longer than the Ada version for small data sets. For large distributions the difference grows quadratically.

3.5 Analysis of results

The different parts of the tool can be used as either scripts or through a graphical user interface depicted in Figure 2. The set of steps to perform is indicated as a toolbar at the top of the screen. The log of the output of the different phases is recorded in the log screen. Commands can also be typed in directly at the command prompt at the bottom.

The user first selects the main file of the program to analyse, secondly the program is compiled for the MIPS architecture with the necessary libraries. After compilation the program needs to be analysed. The user may select which

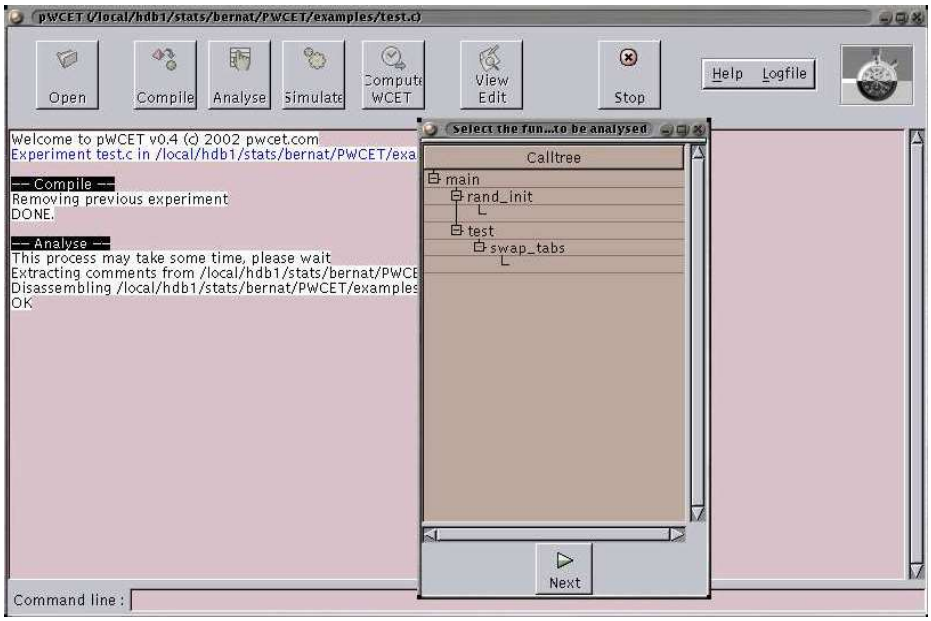

Figure 2: pWCET main window.

functions to include in the analysis. The function selection dialog can be seen in the figure too.

After code analysis traces should be generated. By selecting the simulate option the simplescalar simulator is invoked to run the program. Each run is invoked with a different run number which allows to set up a seed for random number generation, for instance. The parameters that determine the configuration of the simulator can be changed in the pWCET configuration file. This enables the evaluation of the effect that particular processor features have on the execution time of the program. The trace generation also performs the trace analysis by merging the results with previous processed traces.

After the code is analysed and traces generated the XST can be browsed using the code browser, shown in Figure 3. The browser allows to select which function to display. It displays a tree of the selected function. Different types of nodes are indicated by different colors. Each node has information of the type, source line and rule for the timing program generator. The same figure shows the screen that allows the modification of the evaluation rule for a loop node. Several operations can be performed for each node, displaying the source code corresponding to the node, the textual representation of the execution profiles, as well as the graphic plot of the distribution of probability of the node. The graph can show both measured and computed distributions.

The final stage is to launch the timing program generation and calculation.

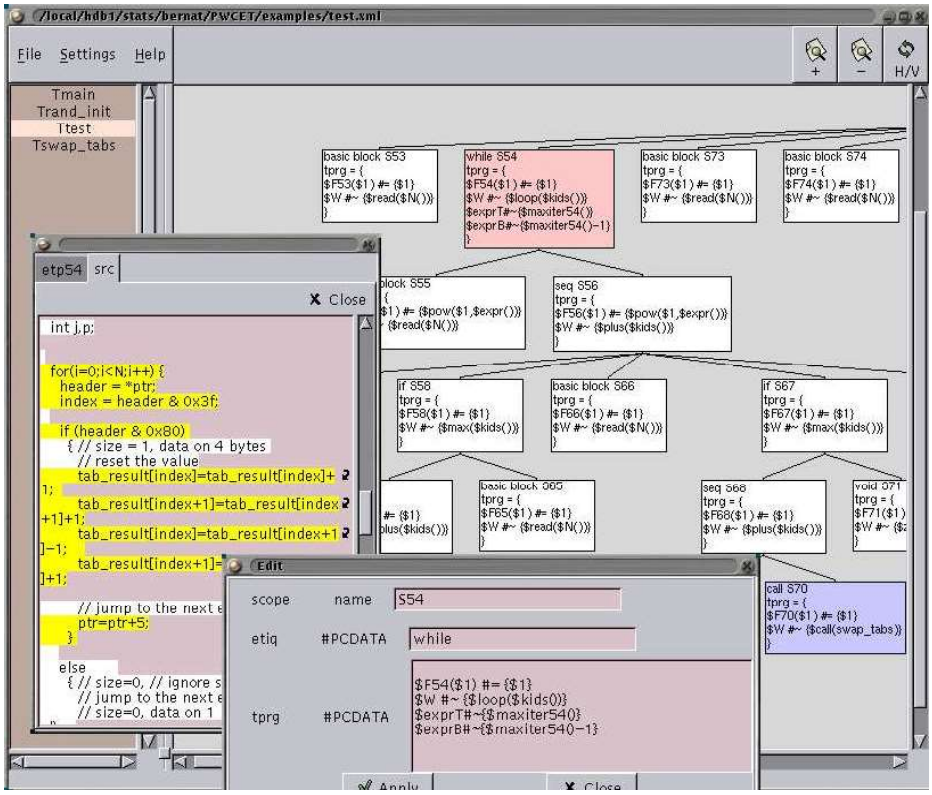

Figure 3: XST Browser showing information window of a node, some graphs and textual output.

The program performs the postorder tree traversal, extracts the rule for performing the WCET calculation from the node attributes section and generates the corresponding program to perform the calculation. The program is then executed by invoking the calculation engine.

After the timing program has been generated and executed, the computed distributions can be viewed with the XST browser. For example when plotting the profiles, both measured and computed profiles are displayed. Examples of such visualisation are shown in the next section.

4 Evaluation

In this section we illustrate the operation of the tool with an example. The program is an implementation of a message processing system. It takes packets from array ptr and decodes them. The type of message and the sign of the data part is encoded in the header. Depending on the different configurations he message is either stored in array tab1 or tab2. A fragment of the program is shown below (for full listing of the program together with other example programs see the pwcet web page).

```
void test() {
 int i,j,p,index;
 char header;
 char * ptr = (char*)data_stream;
 int  * tab_result = tab1;
 int  * tab_error = tab2;
 char * ptr2;
 char * ptr3;

 for(i=0;i<N;i++) {
   header = *ptr;
   index = header & 0x3f;
   if (header & 0x80) {
       tab_result[index]  =tab_result[index]+1;
       tab_result[index+1]=tab_result[index+1]+1;
       tab_result[index]  =tab_result[index+1]-1;
       tab_result[index+1]=tab_result[index]+1;

       // jump to the next element
       ptr=ptr+5;
     }
   else {
       index = header & 0x3f;
       if (header & 0x40) {
         // is positive
         tab_result[index]=-tab_result[index];
       }
       else {
         // is negative
```

```
          tab_result[index]=+tab_result[index];
        }
        // jump to the next element
        ptr+=2;
    }
  }
}
```

The Syntax tree of the fragment of code is shown in Figure 3. Node 54 is the head of the loop. The figure also shows the fragment of the code that corresponds to node 54 as well as the editing window where the expression to calculate the node can be modified by the user. This description is automatically generated.

The simulation generated 1000 traces. The following is a fragment of one of such traces that shows first three iterations of the loop. Note that the trace only shows execution of basic blocks (not of inner nodes in the tree), the analysis part is then responsible to derive the execution of these other nodes. The format is (timestamp node number)*. The timestamp is the cycle number in which the first instruction of the basic block is fetched. For example, in the first iteration node 66 runs for 83 cycles (34943-34860), however in the second iteration it runs for only 13 cycles (and for the rest of iterations in the loop). This is a common behaviour due to cache effects.

```
34399 53 34490 55 34519 57 34631 60 34692 63 34832 64 34860 66
34943 72 34974 55 34978 57 34995 60 35008 62 35098 64 35103 66
35116 72 35123 55 35127 57 35144 60 35157 63 35179 64 35183 66
35196 72 ...
```

The loop analysis determines that in the worst case loop 54 iterates 31 times (this number is the number of times the header is hit). This is indicated with the following maxiter rule:

```
$maxiter54 #= {31}
```

The timing program generated by the tool is shown below. This is an automatically generated ml program. Each node corresponds to an ml function that invokes in its computation recursively the functions that evaluate the sons of the node. The operation of the node is then performed, saved and control returned to the callee. The Following fragment shows the calculation of node 67. Nodes 69, and 71 are basic blocks and its distribution is read from the measured data. Node 70 is a function call to `swap_tabs`. Node 68 is the convolution of the distributions of node 69 and 70. Node 71 was never executed, and therefore is empty. Node 67 is the maximum (probabilistically) of 68 and 71.

```
(*--------- Node 69 -----------*) let w69 () =
  let result = read "ETP69" in
    write (result,"ETP69");
    result;
```

```
;;
(*--------- Node 70 -----------*) let w70 () =
  let result = (ext_call "swap_tabs") in
    write (result,"ETP70");
    result;
;;
(*--------- Node 68 -----------*) let w68 () =
  let result = conv [(w69());(w70())] in
    write (result,"ETP68");
    result;
;;
(*--------- Node 71 -----------*) let w71 () =
  let result = epzero () in
    write (result,"ETP71");
    result;
;;
(*--------- Node 67 -----------*) let w67 () =
  let result = max ((w68())) ((w71())) in
    write (result,"ETP67");
    result;
;;
```

Figure 4 shows the result of the analysis compared to the end to end measurement. The pWCET estimate is an upper bound on the WCET. The distance between the two estimates comes from the fact that the input data does not correspond to the worst possible sequence of data (this is just random messages). The pWCET builds the equivalent of the worst set of input data and plots the profile.

Generation of 1000 traces took 15 minutes on a Pentium 3 at 500 MHz, the generation and evaluation of the timing program was performed in under a minute. The complexity of the timing programs is not greatly affected by the size of the traces.

5 Conclusion

This paper has outlined the theory for probabilistic timing analysis of real-time programs and described the main components of its tool support. The main features are: portability to analyse programs running on different processors and platforms by processing execution traces obtained either by examining the log of a cycle accurate processor simulator or by observing the real system; flexibility: by allowing users to define the way the timing program is generated and therefore enabling different types of analysis. A small case study illustrates the formats of the files involved and the steps of the analysis.

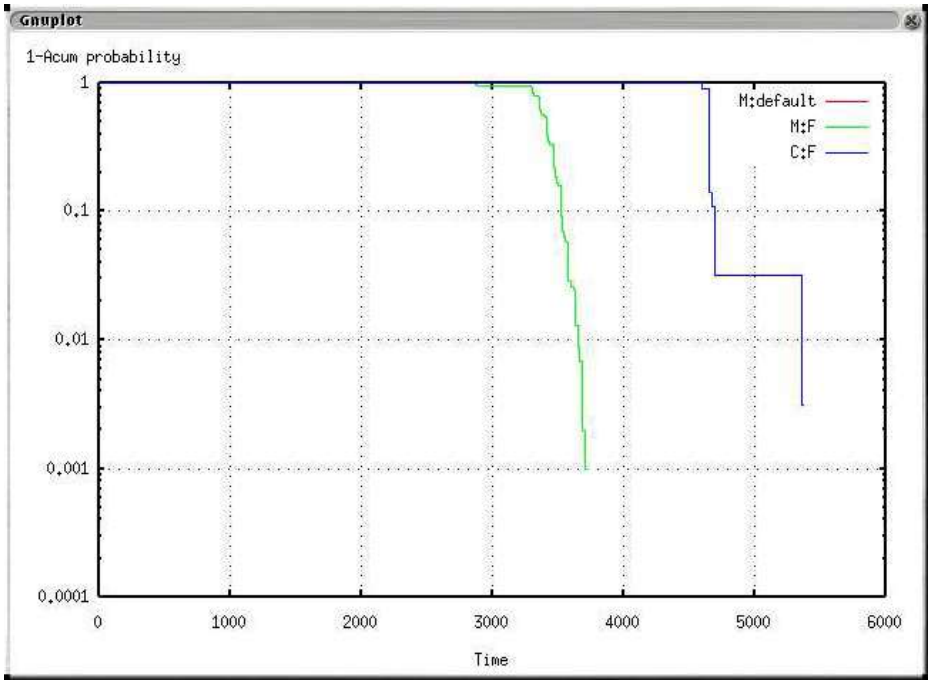

Figure 4: pWCET analysis of node 54. M= Measured, C=Computed. overestimation is due to lack of generating the worst possible input data.

References

[1] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS, Real-Time Systems Symposium*, Austin, TX, USA, December 2002.

[2] Alan Burns and Stewart Edgar. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 19–21 2000.

[3] Alan Burns and Stewart Edgar. Statistical analysis of WCET for scheduling. In *Proc. of the IEEE Real–Time Systems Symposium (RTSS'01)*, London, United Kingdom, December 4–6 2001.

[4] Antoine Colin and Guillem Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 19–21 2002.

[5] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Realtime Systems*, 18:249–274, 2000.

[6] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real–time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Montreal Canada, June 19–20 1998.

[7] Y. A. Liu and G. Gomez. Automatic accurate time–bound analysis for high–level languages. In Frank Müller, Azer Bestravros, et al., editors, *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Lecture Notes in Computer Science, pages 31–40, Montreal Canada, June 19–20 1998. ACM SIGPlAN, Springer–Verlag.

[8] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Realtime Systems*, 17(2/3):183–207, November 1999.

[9] Frank Müller. Timing analysis for instruction caches. *Journal of Realtime Systems*, 18:217–247, 2000.

[10] C.Y. Park and A.P. Shaw. Experiments with a program timing tool based on source–level timing schema. *IEEE Transactions on Computers*, 24(5):48–57, May 1991.

[11] Stefan M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Institute of Real–Time Computer Systems, Technische Universität München, Munich, Germany, 2002.

[12] F. Stappert, A. Ermedahl, and J. Engblohm. Efficient longest executable path search for programs with complex flows and pipeline effects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, pages 132–140, Atlanta, Giorgia, USA, November 16–17 2001.

[13] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by spearated cache and path analysis. *Journal of Realtime Systems*, 18:157–179, 2000.