WCET'2002

2nd International Workshop on Worst-Case Execution Time Analysis

(Satellite Event to ECRTS'02)

Technical University of Vienna, Austria
June 18, 2002

Message from the Workshop Chair

Welcome to the 2nd international Workshop on Worst-Case Execution Time (WCET) Analysis, a satellite event of the Euromicro Conference on Real-Time Systems held in Vienna, Austria, 18th June 2002. This is the second event in the series after the successful first meeting held in Delft.

The aim of the workshop is to provide a forum for discussing current trends and issues related to the timing analysis of Real-Time Systems with special emphasis on bridging the gap between industry and academia. The meeting encourages debate and interaction between participants through short presentations followed by active discussion. The program of the workshop presents contributions on the following areas of timing analysis:

- Within the context of high-level analysis techniques contributions address path analysis techniques and issues related to object oriented programming models.
- On low-level analysis techniques the focus is on modelling timing behaviour of processor features such as cache effects, branch prediction and speculative execution.
- The industrial view presents timing requirements in the aerospace industry and current models of analysis and current tool support.

I would like to express my congratulations to all participants, authors, reviewers, and the organisation committee that have made this event a successful one.

Dr. Guillem Bernat.
University of York. England, UK

Technical Program

- **9:30 - 10:45** High Level Analysis (chair: Isabelle Puaut)
  - *A Prototype Tool for Flow Analysis of C Programs*.
    Jan Gustafsson, Björn Lisper, Nerina Bernmudo, Christer Sandberg and Linus Sjöberg.
    Mälardalen University, Västerås, Sweden.
  - *A novel Gain Time Reclaiming Framework Integrating WCET Analysis for Object-Oriented Real-Time Systems*.
    Erik Yu-Shing Hu, Andy Wellings and Guillem Bernat.
    University of York, United Kingdom.
  - *A Unified Flow Information Language for WCET Analysis*.
    Andreas Ermedahl, Uppsala University, Sweden.
    Jakob Engblom, IAR Systems AB, Sweden.
    Friedhelm Stappert, C-LAB, Germany.
- **10:45 - 11:10** Coffee Break
- **11:10 - 12:00** Tools (chair: Jan Gustafsson)
  - *WCET Estimation from Object Code Implemented in the PERF Environment*.
    Douglas Renaux, João Goés and Robson Linhares.
    Laboratory of Embedded Systems Innovation and Technology, Brasil.
  - *Status of the BOUND-T WCET Tool*.
    Niklas Holsti and Sami Saarinen.
    Space Systems Finland Ltd., Espoo, Finland
- **12:00 - 12:10** Break
- **12:10 - 13:00** Industrial Views (chair: Peter Puschner)
  - *You Can't Control what you Can't Measure, or Why it's Close to Impossible to Guarantee Real-Time Software Performance on a CPU with On-Chip Cache*.
    Nat Hillary and Ken Madsen.
    Applied Microsystems Corp./Wind River Sysrems Inc.
  - *The European Space Agency's Involvement and interest in WCET and Scheduling Analysis*.
    Morter Rytter Nielsen, Eric Conquet and Jean-Loup Terraillon.
    ESA, Nordwjik, Netherlands.
- **13:00 - 14:30** Lunch

- **14:30 - 15:45** Low Level Analysis (chair: Stefan Petters)
  - *Cache Modelling vs Static Cache Locking for Schedulability Analysis in Multi-tasking Real-Time Systems*.
    Isabelle Puaut.
    IRISA, Rennes, France.
  - *A Framework to Model Branch Prediction for WCET Analysis*.
    Tulika Mitra and Abhik Roychoudhury.
    National University of Singapore, Singapore.
  - *Difficulties in computing the WCET for Processors with Speculative Execution*.
    Christine Rochange and Pascal Sainrat.
    Institut de Recherche en Informatique de Toulouse, France.
- **15:45 - 16:15** Coffee Break
- **16:15 - 17:30** Issues in WCET Analysis (chair: Niklas Holsti)
  - *Why You Can't Analyze RTOSs without Considering Applications and Vice Versa*.
    Jörn Schneider.
    Saarland University, Saarbrücken, Germany.
  - *How Much Worst Case is Needed in WCET Estimation?*
    Stefan Petters.
    University of York, United Kingdom.
  - *Is WCET Analysis a Non-Problem? - Towards New Software and Hardware Architectures.*
    Peter Puschner.
    University of Vienna, Austria.
- **20:00** Dinner

Table of content

A Prototype Tool for Flow Analysis of C Programs

Jan Gustafsson, Björn Lisper, Nerina Bermudo, Christer Sandberg, Linus Sjöberg
Department of Computer Engineering
Mälardalen University, Västerås, Sweden
{jgn, blr, nbo, csg}@mdh.se, lsg98020@idt.mdh.se

Abstract

We describe a prototype tool for flow analysis. The purpose of the tool is to statically analyse C programs in intermediate code format, and to calculate flow information, like loop bounds. This information will be used by a subsequent low-level analysis to calculate a final worst case execution time. We describe the main steps of the tool, and analyse a simple example to illustrate our method.

1 Introduction

Predicting the Worst Case Execution Time (WCET) of programs is an essential step in designing real-time systems, especially hard real-time systems. Methods based on static analysis can guarantee the safeness of the predicted WCET, while measurements, in the general case, can not.

In the presence of loops and recursion, finite iteration bounds must be given to the WCET calculation method. Most often, they are given as *manual annotations* by the programmer. Optional annotations (like information on infeasible paths) may also be given, to reduce the overestimation of the calculated WCET. The annotations can be supplied as comments or in a separate file.

A problem with manual annotations is that the calculation of these are often time-consuming and error-prone. It would be advantageous if these annotations could be calculated automatically. This is the aim of the project described in this paper.

The WCET project is a sub-project within CODER (Cluster on Distributed Embedded Real-Time Systems) in the ASTEC [AST01] competence center. The project consists of two groups, one at Uppsala University (low-level analysis) and one at Mälardalen University in Västerås. The flow analysis research is an activity of the Västerås group.

The flow analysis tool is a part of a planned, complete WCET tool (see [EES+01] for details). The flow analysis part will calculate the possible flow of the analysed program. This information will, together with the results from the low-level analysis, be used to calculate a final WCET.

2 Overview of the Tool

2.1 The Input of the Tool

The tool analyzes C-programs in intermediate code format. We will use the NIC (New Intermediate Code) format (developed within CODER). The full ANSI C language will be supported (including pointers, recursion and unstructured code).

We assume that the code represents a syntactically and logically correct program. For example, we assume that array indices are within bounds. We also assume that the control flow graph of the analysed program is the same as in the final machine code, i.e., that the final steps to machine code does not change the control flow.

Manual annotations are used as a complement when the automatic flow analysis fails.

2.2 The Calculation

There will be a possibility to choose between a slower but more exact analysis or a faster but less accurate. It will also be possible to change certain compiler- or system-specific data used in the analysis (like integer type sizes).

2.3 The Output

The purpose of the tool is to calculate flow information ("flow facts", see [EE00]), like number of iterations and recursion levels, infeasible paths etc., that will be used in the subsequent low-level analysis. The flow facts are attached to the scope graph [EE00] of the analysed program. A scope graph is a partition of the program into scopes; a scope is a part of the program where certain flow facts are valid.

2.4 Flow Analysis Overview

Basically, the analysis of a C program is performed using the steps described in Figure 1.

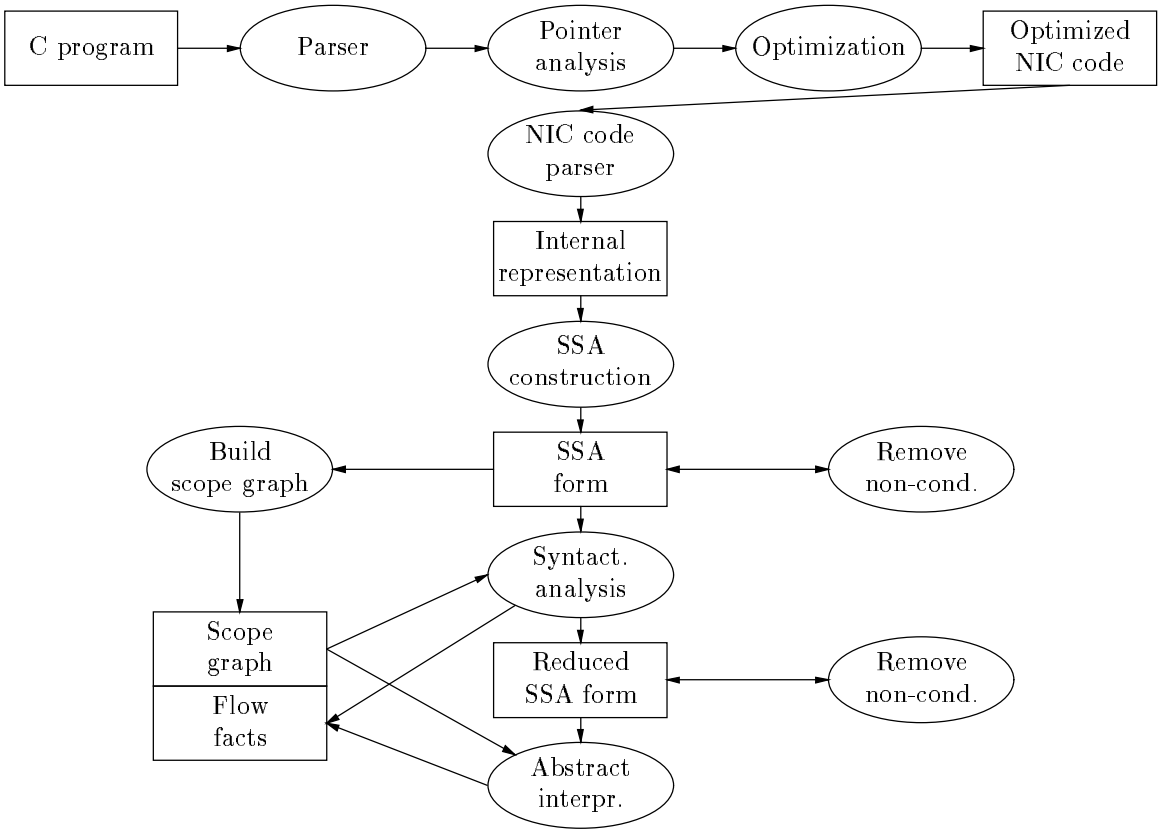


Figure 1: Basic analysis steps.

- Parser. The C code is parsed to produce a NIC file.
- Pointer analysis. Pointers in the program are analyzed. Information about the resulting points-to sets are stored in the NIC file.
- Optimization. The NIC code is optimized. These first three steps are developed within the WPO project.

- NIC code parser. The optimized NIC code is parsed to produce an internal representation. This internal format is the basis for all subsequent analysis steps.
- An SSA (Static Single Assignment) conversion is performed. The calculated data is added to the existing internal representation.
- Non-conditionals are removed. All assignments to variables that do not affect control flow (transitively) are identified and removed from the program. If all references to a variable are removed, the variable will be removed completely. The reason is to simplify and speed up the rest of the analysis.
- Scope graph construction. The scope graph is constructed using the control flow that can be extracted from the internal representation.
- Syntactical analysis. The code is "scanned" for simple, recognizable loop constructs and the corresponding loop counts are calculated, if possible. The loops are replaced with assignments to the final values for the variables updated in the loop, resulting in a simpler program to analyze in the following step.
- The removal of non-conditionals is run again, since variables may become non-conditional during syntactical analysis.
- Abstract interpretation. The remaining code (after the previous step) is analysed using abstract interpretation. The resulting flow facts are appended to the results file.
- If there are constructs for which the abstract interpretation fails, the user is asked for manual annotations for these. The analysis continues with these two last steps until the complete code is successfully analysed.

3 Complete Example

The code in Figure 2 contains a simple and motivating example, activating all the steps of our tool. For simplicity reasons, it does not contain pointers, arrays, or unstructured code. The variable `i` is assumed to receive a value between 0 and 5 by `get_value()`.

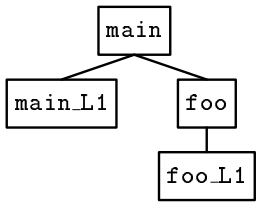
```
int main(void) {
  int i, j, k, n = 10, c = 2, p;
  for (i = get_value(); i <= n; i = i + c) {
    p = i - c * 2; result += 2;
    if (p) k = i + 2;
    }
  j = i + k;
  j = foo(p);
  return(0);
  }
```

```
int foo(int j) {
  int i, result = 0;
  for (i = 0; i < 100; i++) {
    result += 2;
    }
  return(result);
  }
```

Figure 2: Example program

We first parse the code to a NIC file. Conversion to SSA form and removal of non-conditionals (`j`, `k`, `p`, and `result`) yields a NIC code that is equivalent to the C code in Figure 3.

Next step is to calculate the scope hierarchy below. We see that each function and loop constitutes a scope.



```
int main(void) {
  int i, n = 10, c = 2, p;
  for (i = get_value(); i <= n; i = i + c) {
    p = i - c * 2;
    if (p) {}
    }
  foo(0);
  return(0);
  }
```

```
int foo(int j) {
  int i;
  for (i = 0; i < 100; i++) {}
  return(0);
  }
```

Figure 3: Example program after removal of non-conditionals

The syntactical analysis will recognize the loop in `foo` as analyzable and output the flow fact

$$\texttt{foo\_L1:[ ]:}x_{\texttt{header(foo\_L1)}} = 100$$

which means that the loop in scope `foo_L1` iterates exactly 100 times. The notion $x_{\texttt{header(foo\_L1)}}$ refers to the iteration count of the loop header. The function `foo` will be changed by the syntactical analysis as shown below. We see that the loop has been replaced by an assignment.

```
int foo(int j) {
 int i = 100;
 return(0);
 }
```

A new run of removal of non-conditionals removes the variable `i` in `foo` since it does not affect the control flow.

Abstract interpretation of the remaining program will yield the following flow facts for the remaining loop:

1. $\texttt{main\_L1:[ ]:}x_{\texttt{header(main\_L1)}} \geq 3$
2. $\texttt{main\_L1:[ ]:}x_{\texttt{header(main\_L1)}} \leq 6$
3. $\texttt{main\_L1:<4..6>:}x_{\texttt{true}} = 1$

The first two flow facts means that the loop in `main` iterates between 3 and 6 times. The second means that the program will always take the true edge in the if-statement in iterations 4 to 6 of the loop.

References

[AST01] ASTEC (Advanced Software TEChnology) WWW Homepage. URL: `http://www.astec.uu.se/`, November 2001.

[EE00] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[EES+01] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Springer International Journal of Software Tools for Technology Transfer, (STTT)*, 2001.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. 3rd International European Conference on Parallel Processing, (Euro-Par'97), LNCS 1300*, pages 1298–1307, August 1997.

A Novel Gain Time Reclaiming Framework Integrating WCET Analysis for Object-Oriented Real-Time Systems

Erik Yu-Shing Hu*, Andy Wellings and Guillem Bernat

Real-Time Systems Research Group
Department of Computer Science
University of York, York, YO105DD, UK
{erik,andy,bernat}@cs.york.ac.uk

Abstract

This paper proposes a novel gain time reclaiming framework integrating WCET analysis for object-oriented real-time systems in order to provide greater flexibility and without loss of the predicability and efficiency of the whole system. In this paper we present an approach which demonstrates how to improve the utilisation and overall performance of the whole system by reclaiming gain time at run-time. Our approach shows that integrating WCET with gain time reclaiming not only can provide a more flexible environment to develop object-oriented real-time applications, but it also does not necessarily result in unsafe or unpredictable timing analysis.

Keywords : Gain Time, Real-Time Java, Worst-Case Execution Time (WCET) Analysis, Object-Oriented WCET

1. Introduction

There is a trend towards using object-oriented programming languages, such as Java and C++, to develop real-time applications. The success of hard real-time systems, undoubtedly, relies upon their capability of producing functionally correct results within defined timing constraints. In order to achieve this, the processor and resource requirements of the hard real-time tasks have to be reserved. However, this may result in under utilisation and lead to very poor performance for aperiodic tasks. Unfortunately, object-oriented programming languages support more dynamic behaviour than procedural programming languages, and some of these features may bring about object-oriented applications having a more pessimistic worst-case behaviour. In consequence, object-oriented real-time systems may suffer from significantly lower utilisation and poorer overall performance of the whole system than procedural real-time systems.

Most scheduling algorithms assume that the WCET of each task is known prior to doing the schedulability analysis. Typically, the WCET analysis and schedulability analysis are carried out separately. On the one hand, sophisticated techniques for WCET analysis [6, 14, 13], for instance to model caching and pipelining, are used in order to achieve safe and tight WCET estimation. However, most WCET analysis approaches are only considered in relation to procedural programming languages. Some research groups have proposed various approaches [8, 15] to address these issues, but most approaches result in developing environments which are inflexible and very limited. On the other hand, in order to develop more flexible real-time systems, a number of research groups have proposed various flexible scheduling algorithms [5, 12], for instance priority server algorithms [5] and slack stealing algorithm [12]. In general, these flexible scheduling algorithms are mainly focused on the use of WCET to improve the performance of the aperiodic tasks at run-time. They have, however, paid insufficient attention the fact that, for the most part, hard real-time tasks are not executing via the worst-case execution time path. Therefore, even though they have demonstrated very complex scheduling algorithms to improve the average performance of the whole system, the improvements are still limited and the overhead of the implementation is extremely high or it is sometimes not even possible to implement them in practice.

In general, the spare capacity of the real-time system may be divided into three groups [7]: *extra capacity*, *gain time*, and *spare time*. Extra capacity is the capacity which is not allocated for hard real-time tasks during the design phase. This can be identified off-line. The gain time is produced when the hard real-time tasks execute in less than their worst-case execution times. This may only be reclaimed at run-time since it depend on the actual executions of the tasks [7]. The spare time may be defined as a situation in which the sporadic tasks do not arrive at their maximum rate. Most flexible scheduling algorithms are mainly focused on reclaiming the extra capacity of the system. Only some research approaches [9, 1] have discussed how to reclaim the gain time. However, they have tended to focus on procedural programming languages, rather than on object-oriented programming languages.

In this paper we propose an approach which demonstrates how to improve the utilisation and performance of the whole system by reclaiming gain time at run-time. We use a gain time reclaiming mechanism to compensate for the tradeoff among the flexibility, efficiency and predictability. Our approach shows that integrating WCET analysis with gain time reclaiming not only may achieve high utilisation and high per-

*This work has been funded by the EPSRC under award number GR/M94113

formance of the whole real-time system, but also keep the flexibility of the object-oriented real-time applications. The major contributions of this paper are:

- presenting how to address the dynamic behaviour of object-oriented programming features with minimum annotations
- demonstrating how to reclaim the gain times of object-oriented real-time systems with the gain time reclaiming graphs
- balancing the flexibility and predicability of object-oriented real-time applications by integrating WCET analysis

The rest of the paper is organised as follows. Section 2 gives an overview of our previous work. Section 3 demonstrates how gain times can be reclaimed in object-oriented real-time systems. Finally, the conclusion and future work are presented in Section 4.

2. Previous Work

Our previous work, called Extended Real-Time Java (XRTJ)[11], extends the current Real-Time Java architecture [4] proposed by the Real-Time Java Expert Group. The XRTJ architecture has been developed with the whole software development process in mind: from the design phase to run-time phases. For example, using our approach, the system can be evaluated during the design, and the timing constraints of the application can be validated during run-time. We integrate our approach with the portable WCET analysis, proposed by Bernat et al. [3] and extended by Bate et al. [2], for the WCET estimation.

In our previous approach [11], we have introduced the *Extensible Annotations Class* (XAC) format, which stores extra information that cannot be expressed in the source code. The XAC format is an annotation structure that can be stored in files or as an additional code attribute in *Java Class Files* (JCF). We have also addressed dynamic dispatching issues in object-oriented real-time applications [10]. Here, minimum annotations are provided to ensure the predictability of dynamic binding methods and estimate safe and tight WCET for hard real-time applications. However, our previous work mainly focused on the analysis of the hard real-time object-oriented tasks.

3. Gain Time Reclaiming

In order to balance the tradeoff between the flexibility and efficiency of the real-time systems, gain time reclaiming needs to be applied. For the most part, the gain time reclaiming in object-oriented programming languages may be classified in three groups: *structural constraints reclaiming*, *functional constraints reclaiming*, and *object constraints reclaiming*. We use some WCET annotations, which are presented in our previous approaches [11, 10], in figures 1 and 2 . Further details of each reclaiming mechanism are discussed below.

//@ **GainTime**$(Units\ /path\ /mode\ /method)$	–A1
//@ **Dyn_GainTime**$(maxLoopcount, Scope\_Name)$	–A2
//@ **OO_GainTime**$(Object\_Name)$	–A3

Table 1. Gain Time Reclaiming Annotations

Note that the WCET annotations used in the following examples to discuss the gain time reclaiming mechanism can be added either manually by developers or automatically by modified compilers or tools.

3.1. Structural Constraints Reclaiming

From the point of view of the syntax of the programming languages, the real-time tasks allow construction with a number of basic blocks, conditional branches, and call procedures. These components of a real-time task, in general, may be represented by a *control flow graph* (CFG). It can be observed that the actual execution time of real-time tasks may vary, if the execution paths of the task or iteration times are varied at run-time. This section is mainly concerned with reclaiming gain times, which depend on the structural constraints of a specific real-time task.

```
1  ...
2  public check_data () {
3    int i , morecheck, wrongone;
4    i =0; morecheck=1; wrongone=−1;
5
6    //@ DefineScope(checkLoop)
7    while (morecheck) {
8      // Say WCET=20 cycles
9      if (data[ i ] < 0) {
10       //@ GainTime(100 cycles);
11       wrongone=i; morecheck=0;
12       //@ GainTime (Error_mode);
13     }
14     // Say WCET=120 cycles
15     else {
16       ...
17       if (++i >= DATASIZE)
18         morecheck=0;
19     }
20   }
21   //@ Dyn_GainTime(50, checkLoop);
22    ...
23
24   // Say WCET=10 cycles
25   if (wrongone >= 0) {
26     //@ Mode(Error_mode);
27      ...
28     return 0;
29   }
30   // Say WCET=50 cycles
31   else {
32     //@ Mode(Noml_mode);
33      ...
34     return 1;
35   }
36 }
37 ...
```

Figure 1. An example of gain time reclaiming [13]

Our approach is similar to Audsley et al.'s approach [1], which is proposed for procedural programming language. We have defined two annotations ($A1$ & $A2$), which are given in Table 1, to cope with structural constraints reclaiming. As shown in Figure 1, we can annotate the static gain time, such as precalculated unites or paths, with annotation $A1$, and the dynamic gain time, defined for unknown iteration times, with annotation $A2$. In Figure 1, the *if-then-else* basic block can reclaim 100 cycles at Line 10, if the condition expression is TRUE (i.e. $data[i] < 0$) and the *while-loop* is part of its worst case path. With respect to the dynamic gain time, we can simply add an annotation to a non-constant iteration loop, such as *for-loop* or *while-loop*, in order to reclaim gain times at run-time.

Essentially, the gain time can be reclaimed as soon as the

exact execution path of the task or iteration time are identified. Note that either the run-time system, such as the Virtual Machine, must support a mechanism to count the exact iteration of the loop at run-time or addition code must be introduced by an annotation aware compiler to count the loops. It should also be noted that it could be possible that the actual reclaimed gain time is less than the run-time overhead of the reclaiming. In this situation, the gain time should be either neglected or accumulated until it is worth reporting.

3.2. Functional Constraints Reclaiming

This section is mainly concerned with reclaiming the gain times which suffer from functional constraints. This covers the issues that remain from the previous sections, which did not take into account the functional and data dependencies of the exclusive paths or modes of the real-time task.

Identifying the *exclusive paths* [13] or various *modes* [6] in order to calculate the WCET estimation of the real-time program is widely used in the WCET field. Based on design knowledge, the annotations of the exclusive paths or modes may be distinguished during the design phase. Using these annotations, the WCET estimation of each exclusive path or mode may be calculated. However, one should note that it is possible that the WCET estimations of the exclusive paths or different modes are spread over a wide range, and the exact execution path or mode cannot be determined during the design phase. As a result, the WCET estimation could be very pessimistic. In order to address this, we propose a gain time reclaiming framework which takes into account the functional constraints of the structure of the programs.

In our approach, we use the gain time annotation ($A1$), given in Table 1, to identify where the exclusive path or mode can be determined. As soon as the specific execution path or mode is determined or executed, the associated gain time of the executed path or mode can be reclaimed. Again using the previous example in Figure 1, the $A1$ annotation can be annotated at Line 12 to reclaim the functional associated gain time at run-time. It can be observed that using functional constraints reclaiming may reclaim the gain time earlier than the structural constraint reclaiming.

3.3. Object Constraints Reclaiming

So far, we have only discussed the gain time reclaiming which may apply to both procedural and object-oriented programming languages. We have argued for the need to use dynamic dispatching and demonstrated how to guarantee the deadline of hard real-time tasks in our previous work [10]. Our previous approach has shown that allowing the use of dynamic dispatching not only can provide a more flexible way to develop object-oriented hard real-time applications, but it also does not necessarily result in unpredictable timing analysis. Essentially, a *//@maxWCET()* annotation is used to indicate the WCET of a dynamic dispatching method call. However, we cannot avoid the fact that the use of *//@maxWCET()* might have relatively pessimistic results if the class family is too large or the WCET

```
/* ************************
Assume that Class A is a parent
class . Class B, C and D extend A,
and override the m1() methd.
************************ */
...
class App extends RealtimeThread {
  ...
  public void run() {

    //@ OO_GainTime (aa);
    A aa= new A();
    //@ OO_GainTime (bb);
    B bb= new B();
    C cc= new C();
    D dd= new D();
    ...
    /* ************************
     Initial values of x, y and z
     are from the environment.
     ************************ */
    ...
    if (x > 5) {
      ...
      cc = dd;
    }
    ...
    if (y == 5) {
      ...
      aa = dd;
    }
    else {
      ...
      aa = bb;
    }
    ...
    // type changing
    bb = cc;
    ...
    if (z == true) {
      ...
      aa.m1;
      ...
      aa.m1;
    } else {
      ...
      aa.m1;
    }
    ...
    bb.m1;
    ...
    bb.m1;
  }
}
```

Figure 2. An example of object constraints reclaiming

estimations for different classes are spread over a wide range of values. In order to compensate for the penalty of the flexibility of the object-oriented programming, gain time reclaiming is required.

Before discussing further details of the object constraint reclaiming, two novel terminologies are introduced below.

- An *Object Type Lifetime Graph* (OTLG) is a diagram which represents lifetimes of types of particular objects in a specific task. An OTLG is made of two types of component: node and edge. A *node* denotes a place where the type of the object is changed, whereas an *edge* illustrates the lifetime of a particular type of object between two nodes.

- An *Object Gain Time Reclaiming Graph* (OGTRG) is a diagram which illustrates places where the object constraint reclaiming may take place. An OGTRG also consists of two types of component: node and edge. A *node* denotes a place where the gain time can be reclaimed, whereas an *edge* illustrates that there is no gain time reclaiming taking place.

Essentially, the value of the dynamic dispatching gain time of each object can be calculated as follows: *//@GainTime()= //@maxWCET()-//@UseWCET()*. The annotations of the object gain time reclaiming may be generated by using design knowledge or by producing an OGTRG. In order to reduce the run-time overhead, annotation $A3$ may be applied to define which object's gain times are going to be reclaimed. The procedure of object gain time reclaiming is given as follows.

The *control flow graph* (CFG) can be produced from the source code (or Java class file) for each hard real-time task. Based on the CFG, the OTLG, which illustrates the lifetime

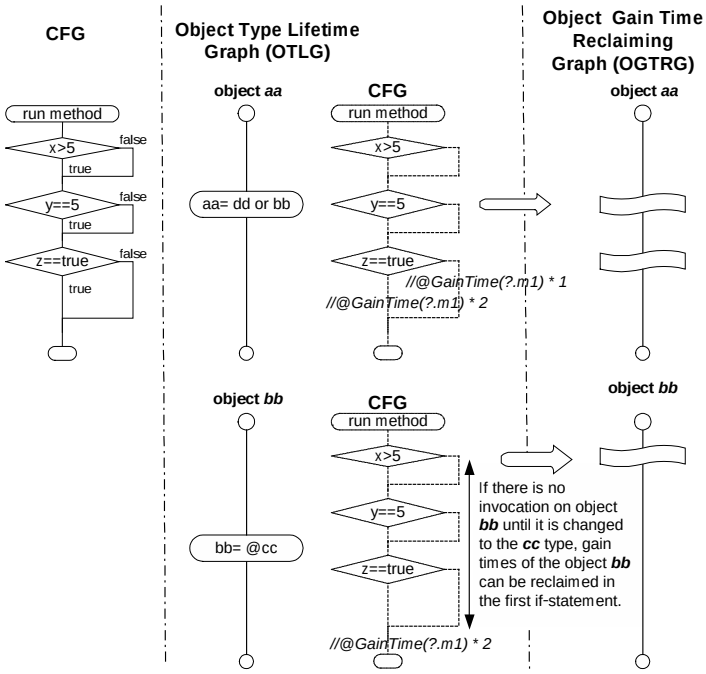


Figure 3. A diagram of producing OGTRG

of an object, can be produced. In the OTLG diagram, symbolic references may be applied to represent the relationship between the dynamic dispatching objects of the same class family during run-time. Using the CFG diagram and the OTLG diagram of each object, the exact places and amounts of gain time reclaiming can be identified. These gain time reclaiming places can be illustrated in an OGTRG for each object. Following this, the gain time reclaiming of all objects in the real-time task can be merged together and provided for the run-time environment (or Java virtual machine) to reclaim them. A diagram which illustrates the transformation from CFG to OGTRG is given in Figure 3.

Solving the symbolic expression of an associated class family can improve the reclaiming as early as possible. As shown in figures 2 and 3, the gain time of the object bb can be reclaimed as soon as the type of the object cc is determined.

4. Conclusion and Future Work

This paper has demonstrated a novel gain time reclaiming framework integrating WCET analysis for object-oriented real-time systems. Our approach shows that integrating WCET with gain time reclaiming not only can provide a more flexible environment to develop object-oriented real-time applications, but may achieve high utilisation and high performance of the whole real-time system.

Here, we have mainly discussed the dynamic behaviour of object-oriented features, which is exclusively restricted to a consideration of the language syntax and semantic aspects. In order to cover as much dynamic behaviour of the object-oriented programming features as possible, our future work has to take into account: memory management, dynamic loading and extension, and remote method invocation (RMI) issues.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems. *In Proc. of the 15th IEEE Real-Time Systems symposium (RTSS)*, pages 12–21, December 1994.

[2] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. *In 6th IEEE Real-Time Computing Systems and Applications (RTCSA2000)*, pages 39–48, December 2000.

[3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. *In proc. 6th Euromicro conference on Real-Time Systems*, pages 81–88, June 2000.

[4] G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *Real-Time Specification for Java*. Addison Wesley, 2000.

[5] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[6] R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. *In Proc. of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[7] R. I. Davis. *On Exploiting Spare Capacity in Hard Real-Time Systems*. Ph.d. thesis, Department of Computer Science, University of York, UK, July 1995.

[8] J. Gustafsson. *Analysing Execution Time of Object-Oriented Programs with Abstract Interpretations*. Ph.d. thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.

[9] D. Haban and K. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *IEEE Transactions on Software Engineering*, 16(12), December 1990.

[10] E. Y.-S. Hu, G. Bernat, and A. J. Wellings. Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 109–116, April 2002.

[11] E. Y.-S. Hu, G. Bernat, and A. J. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 77–84, January 2002.

[12] J. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. *In Proceedings of 13th IEEE of Real-Time Systems Symposium (RTSS)*, pages 110–123, December 1992.

[13] Y. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *ACM SIGPLAN Workshop on Language, Compilers and Tools for Real-Time Systems*, June 1995.

[14] F. Mueller. *Static Cache Simulation and its Applications*. Ph.d thesis, Department of Computer Science, Florida State University, July 1994.

[15] P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, June 2000. St. Malo, France.

A Unified Flow Information Language for WCET Analysis

Andreas Ermedahl[†]
IT-Dept. Uppsala University
Box 337, SE-751 05 Uppsala
Sweden
andreas.ermedahl@it.uu.se

Jakob Engblom[†]
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
jakob.engblom@iar.se

Friedhelm Stappert[*]
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
friedhelm.stappert@c-lab.de

Abstract

In this paper we raise the question if it is possible to create a unified flow information language that all WCET research groups can agree upon, and that is independent of flow analysis and calculation methods.

We discuss desired characteristics of such a flow information language and describe the type of flows that it should be able to express. We present our previously published flow fact annotation language and discuss how it fulfils the desired language properties.

1. Introduction

A correct WCET calculation method must take into account the possible program flow, like loop iterations and function calls. For expressing program flows numerous annotation languages have been presented in the WCET literature. The expressiveness and the type of flows that can be handled by these languages mostly depend on the characteristics of flow analysis methods used, rather than being targeted for the potential WCET tool user.

To generate a WCET estimate, we consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*. Most WCET research groups make a similar division notationally, but sometimes integrate two or more of the phases into a single algorithm.

The program flow analysis phase determines possible program flows, and provides information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. The information can be obtained by *manual annotations* (integrated in the programming language [14] or provided separately [6, 9, 19]). The flow information can also be derived using *automatic flow analysis* methods [7, 10, 13, 22].

In the calculation phase a program WCET estimate is derived, combining the information derived in the program flow and low-level analysis phases. There are three main categories of calculation methods proposed in literature: *tree-based*, *path-based*, and *IPET* (Implicit Path Enumeration Technique).

In a *tree-based* approach the WCET is calculated in a bottom-up traversal of a tree generally corresponding to a syntactical parse tree of the program, using rules defined for each type of compound program statement (like a loop or an `if`-statement) to determine the execution time at each level of the tree [1, 2, 16, 20].

In a *path-based* approach the possible execution paths of a program or piece of a program are explored explicitly to find the longest path [10, 12, 22, 23]. The path-based approach is natural within a single loop iteration or function.

In *IPET*, program flow and low-level execution time are modeled using arithmetic constraints [6, 9, 15, 18, 21]. Each basic block and program flow edge in the program is given a time (t_{entity}) and a count variable (x_{entity}), and the goal is to maximize the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows.

2. Representing Program Flow

The program flow phase can be further divided into three different subphases:

1. **Flow analysis**: Obtaining flow information. By manual annotations or automatic flow analysis.
2. **Flow representation**: Representing the results of the flow analysis.

[†] This work is performed within the Advanced Software Technology (ASTEC, `http://www.docs.uu.se/astec`) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, `http://www.nutek.se`).

[*] Friedhelm is a PhD student at C-LAB (`www.c-lab.de`), which is a cooperation of Paderborn University and Siemens.

3. **Calculation**: Using the control flow information (as represented in the flow representation) in the final WCET calculation.

Some WCET methods integrate two or more of the phases. We believe that the separation of the flow analysis from the calculation reduces the complexity of each stage. Also, by keeping the flow analysis phase separate from the flow representation, results from several different flow analysis methods and manual annotations can be integrated and used together in the calculation phase.

When designing a language for expressing flow information there are a number of choices to be made:

- *Expressiveness:* What type of flows should be possible to express? What type of language constructs should be used?
- *Code relation:* How is the information related to different entities in the program code?
- *Calculation conversion:* How should the information be used in the final calculation phase?

2.1. Expressiveness

We first note, that a natural way to give flow information is by constraining the number of times different program entities, e.g. loops, statement, nodes or edges, can be taken. This can either be precise bounds, e.g. that a loop is iterated exactly ten times, or upper or lower bounds, e.g. that node `A` can't be taken more than five times. It is also beneficial if we can relate the executions of different program entities, e.g. that node `A` and node `B` will always be executed together.

The language can consist of named special relations between entities (e.g. using constructs like Parks `samepath(A,B)` and `nopath(A,B)` [19]). An alternative is to use a more generic style based on math, like our flow fact language [6]. The benefit of a generic math-based language is that it can express flows that are hard to put in words and that there is no obvious limit to the types of flows that can be expressed. On the other hand, a special purpose language is easier to understand, but requires that new language constructs are invented in order to express new flows.

The language must reflect the flows found in real-world programs. Researchers have investigated embedded software [4], the RTEMS operating system [3] and common signal-processing algorithms [8]. The results are not in complete agreement on the properties and flows typical for embedded software, showing that more research and knowledge is needed here.

One observation is that flow information is mostly *local* in its nature, specifying something valid for a small part of a program or a particular invocation of a function. Thus, it is not always suitable to specify flow information once for each entity in the program. E.g. we would like to be able to specify that some node `A` can't be executed during the first five iterations of a loop or give a loop bound valid for just some particular executions of a loop. A language should allow for such local flow information to be expressed.

2.2. Code Relation

First we note that it is natural to express flow information in relation to the entities available in the program code. Flow information can be provided in relation to the source code, intermediate code in a compiler, or the object code. If provided on source code level, the information must be mapped to the object code to be used in the WCET calculation. In the presence of optimizing compilers, this problem is non-trivial [5, 17].

Automatic flow analysis is probably easier to perform at the source code or intermediate code, since variables and other entities of interest are harder to identify in optimized object code. Also, for the potential WCET-tool end-user manual annotations are typically easier to provide at the source-code level.

Another issue is if the flow information should be included as a part of the programming language or provided outside the program. The benefit of language inclusion is that it forces the programmer to write code in an analysable manner. However, this requires compiler support and makes it harder to try different scenarios.

Specifying the flow information outside the program source allows it to free itself from the static structure of the program. For example, by using a *call-graph* representation, we can differ between invocations of the same function when called from different places in the code. An example of the extended version is our *scope graph* represention [6].

A good language should provide *stability* in that program changes not related to annotated code should not force the annotations to change. For example, a problem with expressing flow information on the object code level is that the information might need to be regenerated every time the program code changes.

An important issue is the ability to handle *unstructured code*, e.g. due to uses of `goto` and jumps into loops. An optimizing compiler might produce unstructured object code from structured source code, and automatic code for state machines also tends to be unstructured. A general purpose flow information language must be general enough to express flows over such unstructured code.

2.3. Calculation Conversion

Regardless of the flow information language used the extracted flow information must be "compiled" or

```
if(i < 10) A;   // Stmt B and C
else B;         // can not be
if(i <= 7) C;   // taken together
else D;
```
(a) Infeasible path

```
for(i=0;i<10;i++)     // bound: 10
  for(j=i;j<10;j++)   // local bound: 10
    E;                // E executed at
                      // most 55 times
```
(b) Triangular loop

```
if( cond )
  x = true;   // stmt: F
for( ... )    // Execution of G
  if( x ) G;  // is implied by F
```
(c) Deeply nested dependency

Figure 1. Example of Code with Different Type of Flows

"adapted" to the calculation method used. The adaptation must be *safe*: never exclude execution paths which are considered possible by the flow information, and *tight*: including as few extra execution paths compared to the provided flow information. Figure 1 gives example code showing that not all calculation methods can take advantage of all types of flow information.

The tree-based method [1, 2, 16, 20] is conceptually simple and computationally cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependencies between statements. For example, the code and flow information in Figure 1(a) causes problems in a tree-based calculation method since the timing of the first if-statement will be calculated in isolation from the second if-statement.

The path-based approach is natural within a single loop iteration or other executions of one loop [11, 23]. The method has problems with flow information stretching over loop borders and/or flow information on the *total* number of times entities are taken. For example, the path-based method has problems handling the "triangular" loop dependency in Figure 1(b). If WCET calculation is performed locally, the WCET calculation for the inner loop will assume 10 iterations, and the WCET calculation for the outer loop will use 10 executions of the inner loop, leading to the body of the inner loop being counted 100 times, when it is actually never executed more than 55 times.

For IPET very complex flows can be expressed using constraints, but all flow information needs to be given on a global program level [6, 9, 15, 18, 21]. This contradicts the need to specify flow information in a local context. As shown in [6], local flows can be handled by unrolling the program and lifting the information to a global level. Since flow information is given as relations over count variables some type of flow implications are problematic to express. E.g. Figure 1(c) shows an example of code where we would like to express an implication dependency like: "if `F` is taken once then (and only then) `G` can be taken several times, but if `F` is not taken then `G` can not be taken either".

3. Our Flow Fact Language

This chapter describes our previously published flow fact annotation language [6] and discusses how it fulfils the desired language properties.

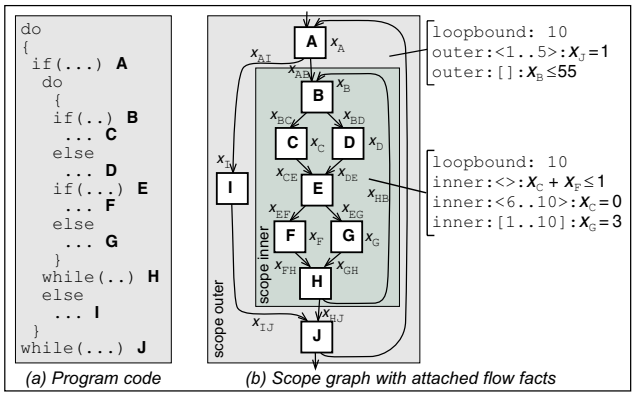


Figure 2. Scopes with Attached Flow Facts

The program representation used is the *scope graph*. It is a hierarchical representation of the dynamic structure of the program. Each *scope* corresponds to a certain repeating or differentiating execution context in the program, e.g. loops and function calls, and describes the execution of the object code of the program within that context. Figure 2(b) shows the scope graph generated for the code in Figure 2(a).

A scope consists of a number of *nodes* and *edges*. A node belongs to exactly one scope, and represents the execution of a certain basic block in the program in the environment given by the scope and its ancestors. For each scope, a header node must be given. If the scope iterates, each iteration must pass the header node, and a bound on the number of iterations has to be provided.

To express more complex program flow information than just basic loop bounds each scope can carry a set of *flow facts* [6]. The flow facts use constraints local to a scope to describe the flow. The constraints can be given for a range of iterations, or all iterations of a certain loop. They can also be local within a single iteration ("foreach facts") or represent a total over all iterations ("total facts").

The scope graph in Figure 2(b) has been decorated with some flow facts.

Flow fact `inner:<>:`$x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$ is a foreach fact and gives that the nodes `C` and `F` cannot be executed on the same iteration of the scope `inner` (an infeasible path), while the flow fact `inner:<6..10>:`$x_{\mathtt{C}} = 0$ gives that for each entry of `inner`, during iterations 6 to 10 of `inner`, node `C` can not be executed.

Flow fact `inner:[1..10]:`$x_{\mathtt{G}} = 3$ is a total fact that gives that, for each entry of `inner`, during the ten first

iterations, node `G` must be taken exactly three times.

Compared to the criteria given above, we note that the flow facts language uses the math-based style and allows us to give local information. The information is given outside the code and uses an expanded version of the call graph (and thus the control flow graph). In its current version, it cannot handle all types of unstructured code due to the need for a header, and since it relates to the object code, it is very sensitive to program changes.

It has been used to perform both IPET- and path-based calculations [6, 23], but not all facts could be used in the path-based approach. It is interesting that the path-based calculation recognized certain types of facts as meaning "samepath" or "not samepath", and exploited these by rewriting the graph.

References

[1] R. Chapman. Program Timing Analysis. Dependable Computing System Centre, University of York, England, May 1994.

[2] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, May 2000.

[3] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[4] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[5] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10th Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[6] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[7] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[8] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[10] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[11] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[12] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, pages 79–88, June 1999.

[13] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[14] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[15] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[16] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[17] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

[18] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[19] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[20] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[21] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[22] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[23] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems, (CASES 2001)*, November 2001.

WCET Estimation from Object Code implemented in the PERF Environment

Douglas Renaux, João Góes, Robson Linhares
Laboratory of Embedded Systems Innovation and Technology – LIT
CEFET-PR
Brasil

Abstract

"*The estimation of the Worst Case Execution Time of a function produces results that are **safe** and that have a **low error**, even in architectures using pipelines and caches.*" This is our thesis; in this paper we present results that indicate that this thesis is correct.

The two basic approaches to obtain WCET of a piece of code are estimation and measurement. At LIT, a tool called PERF is under development. This tool uses both approaches so to obtain the best of both worlds. Measurement provides precise results, but requires the target to be built and running the worst possible scenario, which is often hard to determine. On the other hand, the precision of estimation methods is highly dependent on the complexity of the estimation model. PERF is a design and evaluation environment: a project can be defined, files can be edited, compiled, linked; the resulting code can be analyzed from a timing perspective both via estimation and via measurement. In this way, we intend to encourage the developer to perform time estimations as early as possible in the design cycle. Any tool (commercial or academic) can be inserted in PERF via *plug-ins*. This was the case of the text editor, the compiler and the linker. Hence, PERF is actually a framework to which many tools can be added. PERF works with the object code generated by the integrated tools, in order to obtain execution time limit estimations for functions that compose a real-time systems' software project.

In this paper we present the PERF environment's architecture, with emphasis on the integrated time estimation model and the results obtained using this model.

1. Introduction

Any tool which intends to estimate execution times should take two different domains into consideration: the source code, which the developer usually uses to develop his software project, and the executable code, on which the time estimations are actually performed.

The problem of execution time estimation of a program is usually divided into 3 sub-problems: execution path analysis on source level; source code and machine code correlation; and execution time analysis for each individual machine instruction at each path of the object code.

A strategy based on these 3 sub-problems would search for existing paths in the source code and tries to relate them with the corresponding paths in the object code. Meanwhile, this relationship may not be easily determined, especially in cases where the code optimizations are enabled. The correlation between source-code and object-code may not be trivial, specially for a tool which intends to do this automatically; so, it is possible to concentrate the estimation tool's work on the object-code path's determination and on the estimation of each of its execution times, leaving to the compiler the responsibility of doing the correlations and interpreting the results obtained through the analysis of object code. Another advantage of this strategy comes from the possibility of analyzing object-code present in code libraries, to which the source code is usually not available.

The resolution of the third sub-problem, concerning to the individual execution times of any machine instruction, is affected by the quality of the model that expresses the hardware platform on which the code is to be executed. In order to minimize the estimation error, when compared to the measured time values for a given execution path, this model should consider internal architectural features that have any influence over the execution times of the instructions, such as cache memories and pipelines (if available). Moreover, it is useful for the model to be reconfigurable, so that the estimation tool is able to address several target architectures of real-time systems. The reconfigurability feature can be available to the tool's user, in a way that allows him to adapt the estimation process to the architectures of his interest, using configuration parameters; that requires the estimation algorithms to be generic and able to use the configuration parameters of any architecture in a similar way.

2. The PERF environment

PERF is a tool, under development at LIT, which intends to be a complete design and evaluation environment. At PERF a software project can be defined, edited, each of its modules can be compiled and linked; moreover, the resulting object code can be analyzed from a timing perspective both via estimation and via measurement, which makes PERF suitable for development of software for real-time systems. In this way, the use of PERF intends to encourage the developer to perform time estimations as early as possible in the design cycle.

PERF's architecture is composed of a central core, controlled through a graphical interface. This core is intended to manage a set of integrated tools, each one performing a determined task in the development process (editors, compilers, linkers, time analyzers, etc.) and configured via a *plug-in*. Any tool (commercial or academic) can be inserted in PERF via *plug-ins*, allowing the developer to use only the functionalities which are strictly needed for the target platform and the type of design /evaluation process of interest.

2.1 Time estimation tool

PERF aggregates an execution time estimation tool, whose estimation process is shown in Figure 1. This figure shows that an important design decision of PERF's estimation tool was to analyze object code, instead of source code or executable code. Object code has all the information that is relevant for timing analysis and it may include debug information that relates the executable code to the source code. Also, by analyzing object code, all the libraries can have their timing information extracted, even the commercial libraries.

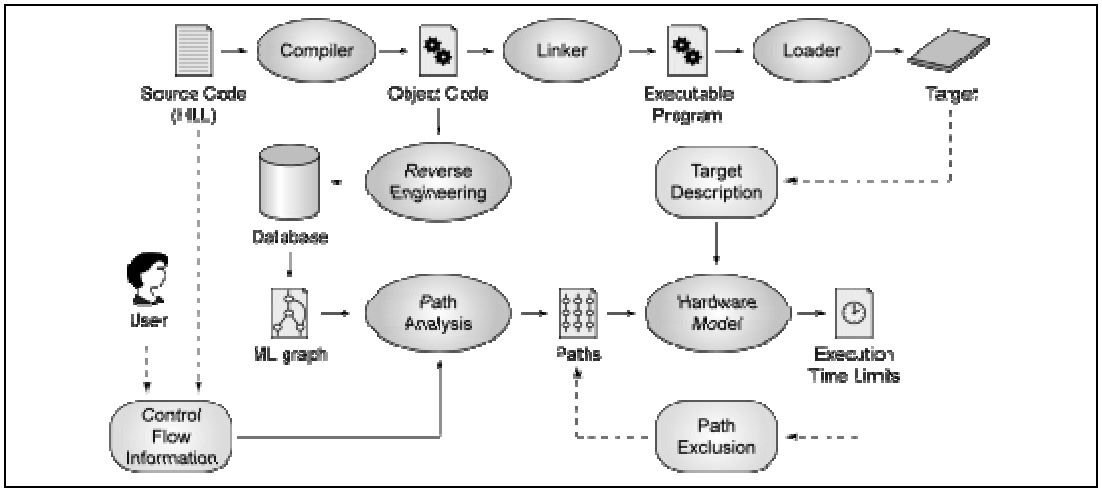


Figure 1 - Time estimation process used in Perf

Figure 2 shows how PERF presents the different views of a function to the developer. It is worth noticing that the correlation between source and machine code can also be obtained through debug information (as it is shown by the gray shaded areas on the source code, machine code and control flow graph views); nevertheless, the developer can still be asked to provide data flow information necessary for the estimation, such as number of loop iterations, so that the several execution paths found in the control flow graph (CFG) are correctly estimated and the correct values for BCET, TCET and WCET are shown in their respective view.

How does PERF obtain execution time estimations (Figure 1):

1) Analyze the object code and extract the control flow graph.

Each node of the graph represents a code segment, i.e., a sequence of machine instructions at which only the last instruction can be a branch (absolute jump, conditional jump, procedure call, software interrupt,...) and only the first instruction is the target of a branch. The control flow graph is obtained in a

function-by-function fashion, even for the commercial libraries, and each function can have its CFG individually shown, as well as the possible correlation with the source code (if available).

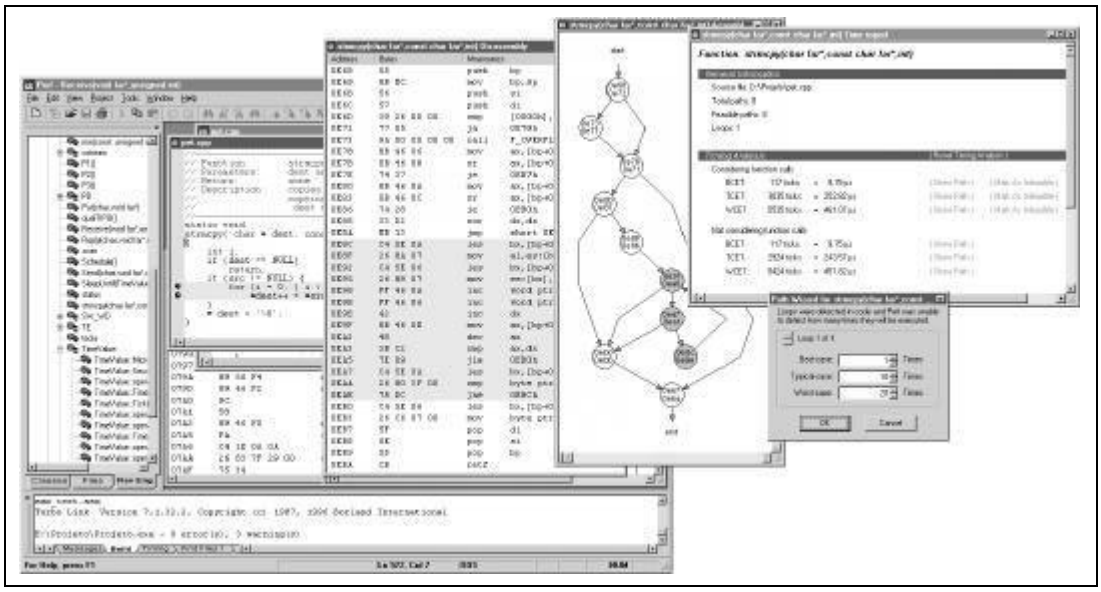


Figure 2 - The PERF tool presenting several views of a function: source, machine code, control graph and timing information

2) All possible paths in a procedure call are analyzed.

Loop structures require information from the user specifying the minimum, typical, and maximum number of iterations. This information can be given in the form of comments in the source code or requested interactively during the analysis (Figure 2).

3) For each path, a time estimation (BCET, TCET and WCET) is obtained based on the configured hardware model for the target platform.

The hardware model is the main functionality of the execution time estimation tool. This model intends to address the third sub-problem of the execution time estimation problem (the analysis of the individual execution times for each machine instruction) by considering internal architectural features of the target platform's hardware.

The computation model implemented at PERF is divided in two basic parts: the estimation algorithms, which are generic and should work in a similar view, independent of the target platform; and the architecture configuration parameters, obtained from an external *plugin.*

The generic estimation algorithms consider the influence, on machine instruction execution times, of multi-stage pipelines, instruction caches of several sizes and associativity degrees and prefetch queues. It is possible, considering the influence of these features in a correct way, to obtain time estimation values with estimation errors lower than 10%, comparing to measured time values; this is valid not only for RISC architectures but for CISC architectures too. The efficiency of the model's work depends on a correct configuration for the architectural parameters via the configuration plugin; it is conceived as a series of data structures in C++ programming language, which can be configured by a developer using the manufacturer information about the target platform.

The model differentiates the values of BCET and WCET, for each analyzed path, by a pessimistic factor introduced in the algorithms. For this factor to be correctly addressed, the best and worst case execution time values for each machine instruction should be modeled during the configuration of the model; moreover, pessimistic and non-pessimistic considerations are also made concerning to the removal of instructions from the prefetch queue and the existence of bus conflicts between instruction and data accesses.

TCET values, by their way, are obtained through typical-case annotation, provided by the user for the number of loop iterations; in functions where no loop structure is present, the TCET value is the same as the WCET value.

The loop processing, for any analyzed path, is done using a strategy that eliminates loop redundancies. This decreases the processing times for loops with a great number of iterations.

4) A report is generated to inform BCET, TCET and WCET of each function.

The execution time of called functions is included in the execution time of the caller, as long as the execution time of the callee is known. Several iterations may be required to evaluate all these dependencies.

3. Results

Two processor architectures were initially considered for the time estimation process and modeled, using the proposed hardware model. The first one was a Intel 80C186EC processor, a CISC core with two pseudo-pipelined stages, a 6-byte prefetch queue and no instruction cache memory; the second one was a Motorola PowerPC MPC860 processor, a RISC core with an 8-stage pipeline and a 2-set associative instruction cache memory of 4 Kb.

The estimation tool was used to estimate the execution time of about a hundred functions, including all the functions of a real-time kernel. The results obtained from the estimation tool are all safe (the estimated WCET is never lower than the actual) and the maximum error was of 10% for the 80C186EC architecture and 15% for the PowerPC architecture. This error, above the initial requirement of 10%, is due to the inadequate equipment used to validate the model by comparison with measured values; this is one of the difficulties that can be faced by the developer which intends to configure his own processor model, provided that the manufacturer information for each instruction's execution time of a certain processor are not always accurate.

Some weaknesses are still detected on PERF 's execution time estimation tool:

- The inefficiency of solving all kinds of jump tables, used for indirect branch operations (for example, the compilation of most 'switch' structures). PERF is only capable at this moment to find jump tables in architectures for which the base address is encoded in the instruction code (for example, the Intel 80C186EC architecture).
- For optimized code and code for which the source is not available (for example, libraries), it can be difficult for the developer to determine the exact number of some loops' iterations, when asked by the tool to provide those values.

4. Conclusions and future work

On going work in the PERF environment include: improving the computational models for the PowerPC architecture to reduce the estimation error, development of computational models of data caches (currently only instruction caches are modeled) and a tool for scheduling analysis is also under development.

The results obtained so far with PERF are very encouraging. Our previous work on timing analysis included the analysis of source code, of the assembly listing produced by the compiler and of binary code. Each one has strong aspects, but from our perspective, object code analysis is the best option.

References

RENAUX, D. P. B. and DASIEWICZ, P. **RTX-Parlog: Real-Time Extended Parlog**. In Euromicro'93 Workshop on Real-Time Systems, p. 147-153. IEEE Computer Society Press, June 1993.

GÓES, João Alexandre. PERF: **Ambiente de Desenvolvimento e Estimação Temporal de Sistemas em Tempo Real**. Dissertação de Mestrado - CEFET-PR / CPGEI - Julho / 2001

LINHARES, R. **Modelamento de Hardware Visando À Estimação do Tempo de Execução de Programas**. Dissertação de Mestrado, CPGEI, CEFET-PR. Curitiba, PR, Dezembro 2001

Status of the Bound-T WCET Tool

Niklas Holsti and Sami Saarinen
Space Systems Finland Ltd
Niklas.Holsti@ssf.fi, Sami.Saarinen@ssf.fi

Abstract

Bound-T is a tool for static WCET analysis from binary executable code. We describe the general structure of the tool and some specific difficulties met in the analysis of the supported processors, which are the Intel 8051 8-bit microcontrollers, the Analog Devices ADSP-21020 Digital Signal Processor, and the SPARC V7 processor. For the DSP, the problem is the complex program sequencing logic using an instruction pipe-line and nested zero-overhead loops with implicit counters and branches. The solution is to model the full sequencing state in the control-flow graph. For the SPARC, the problems are the register-file overflow and underflow traps, which may occur at calls and returns, and the concurrency of integer and floating-point operations, which may force the Integer Unit to wait when it interacts with the Floating-Point Unit. The traps are modelled with a whole-program analysis. The IU/FPU concurrency is modelled by distributing the potential waiting times onto flow-graph edges in a heuristically optimal way, also using some inter-procedural analysis.

1 Introduction

The Bound-T tool analyses compiled and linked executables to find the WCET, flow graphs, call graphs, and stack usage. Space Systems Finland (SSF) developed the tool with support from the European Space Agency (ESA) for space applications. SSF is developing the tool further, aiming also at non-space applications.

The target processors currently supported are the Analog Devices ADSP-21020 (a 32-bit floating-point DSP architecture, forerunner of the SHARC), the Intel 8051 (a large family of 8-bit microcontrollers), and the SPARC V7 (a 32-bit RISC general-purpose architecture). All these processors are used in ESA space projects, which is one reason why they were chosen for Bound-T. The other reason was to implement some quite different architectures in order to verify the adaptability of the tool design.

The most advanced feature of Bound-T is the automatic analysis of loop-bounds. The same analysis provides some context-sensitivity by propagating actual parameter values into the analysis of the called subprogram. To supplement the automatic analysis, the user may state assertions on loop bounds, variable ranges, and other useful facts.

The main limitations of Bound-T are currently the lack of analysis of cache memories, a limited analysis of aliasing and dynamic branching, and the difficulty of high-level analysis based on low-level code, especially if the machine word is short, for example a loop with a 16-bit counter running on an 8-bit machine. The arithmetic analysis is occasionally very time-consuming and sensitive to the structure of the program.

The rest of this paper paper is organized as follows. Section 2 discusses the architecture of the tool. Section 3 describes the modelling of the target processor and target program thru abstract data types. Section 4 presents the major analysis phases and methods. Sections 5 and 6 focus on some interesting and difficult problems: the control-flow analysis of the ADSP-21020, and the IU/FPU concurrency in the SPARC. Section 7 reports on the commercialization and section 6 sketches future work.

2 Tool architecture

Bound-T is based on target-specific modules for reading and decoding binary files, generic modules for creating the control-flow graphs and call-graphs, a Presburger Arithmetic package (*Omega*) for modelling the arithmetic of loop-counters, and an Integer Linear Programming tool (*lp_solve*) to find the worst-case path.

The architecture of Bound-T was designed to be *adaptable* to different target processors, *extensible* with new kinds and methods of analysis, and *portable* to different host platforms.

The easy part of adaptability is to isolate the target-dependent parts into target-specific modules. The hard part is to make the interface of these modules valid for all targets. Our approach is to abstract the important aspects of the target processor. This has worked well, as shown by the range of supported targets.

Extensibility is provided in the conventional way by dividing the analysis into phases, with the result of each phase stored in the program-model. This method is limited by the fact that the data structures of the program model are hard-coded (as opposed to a data-base, for example). However, there are hooks to target-specific data and operations which have let us implement the new SPARC analyses in the Bound-T framework.

For portability, we use a portable implementation language (Ada). The current user interface, based on the command-line and text inputs, is trivially portable.

3 Processor and program models

3.1 Processor model

The target processor is modelled by several abstract data types. The most important type is the identifier or *address* of a control-flow *step*. For a simple processor like the Intel 8051, a step-address is just the address of an instruction. For a processor with complex program-sequencing, a step-address can contain much more context (see section 5 for the ADSP-21020 example). The step-address type is the basis for creating the control-flow graphs, where nodes are identified by a step-address.

Another important abstract type models the registers, flags and memories of the processor, or in general any *cell* that can store an integer value. The cell type is the basis for modelling the arithmetic computations and branching conditions. Some cells are just an enumeration, for example all the processor registers that are always statically addressed. Other cells can represent storage addressed in complex or dynamic ways, for example parameters accessed relative to the stack pointer.

These two abstract types divide our model of program state into a *control state* (step-address), modelled by the control-flow graph, and a *data state* (values of cells), modelled by Presburger input-output relations.

Our processor model is essentially limited to single-threaded processors, although synchronized internal concurrency is possible. In other words, there may be several functional units running concurrently, as long as they all execute the same instruction stream as in VLIW machines. In the SPARC, the IU and FPU are not that strictly synchronized, and so this processor is in principle out of scope for our model.

3.2 Subprogram model

A subprogram under analysis is represented by a control-flow graph (CFG). A node in the CFG is a basic block and contains a sequence of steps. A step usually corresponds to a machine instruction, but in special cases instructions may be split into several steps, or consecutive instructions may be bundled into one step. For example, the Intel-8051 needs two consecutive 8-bit immediate-load instructions to load an immediate 16-bit value into the 16-bit Data Pointer register, but the arithmetic analysis is easier if the two 8-bit instructions are decoded into one 16-bit step in the CFG.

Calls to other subprograms are represented by special CFG nodes (steps) that refer to the callee.

3.3 Timing model

Each step in a CFG has an associated worst-case execution *effort* which depends mainly on the instruction(s) the step represents, but can also depend on the context (via the step address). The effort is a target-specific abstract type, as are the types for the total *work* to execute a sequence of steps and edges, and the processor *power* that determines the number of processor cycles taken by some amount of work. Memory accesses and wait-states are modelled in the effort, work, and power.

Each edge in a CFG has an associated worst-case execution time which typically models two things: (1) the extra time taken to actually branch from a conditional branch instruction, and (2) interference between instructions. For example, when an ADSP-21020 instruction that uses an address generator is immediately preceded by a load-register into this address generator, it takes two cycles instead of one cycle. We assign the extra cycle to the edge between the two instructions.

Branch delays due to instruction pipe-lining, where a branch takes effect only some "delay slot" cycles, are modelled in the step-address (sequencer model), not in the execution time of the branch edge.

Since we use ILP to find the worst case path in a CFG, we assume that the total execution time of a path is at most the sum of the times for the nodes and edges on the path.

3.4 Arithmetic model

The arithmetic effect of a step is represented by a set of assignments of expressions to cells. The expressions are Presburger formulas, possibly conditional, operating on constants and cell values. It is also possible to state that a cell is set to an unknown value. For example, consider a step (an instruction) that increments the register A and sets the Z flag if the result is zero. The effect is represented by the two assignments $A' = A + 1$ and $Z' = \mathit{if}\ A+1 = 0\ \mathit{then}\ 1\ \mathit{else}\ 0$, where a prime indicates the new value of a cell, so A' = new value of A.

The Presburger formalism could in fact model any Presburger relationship between the "before" and "after" values of the cells, for example $A = A' - 1$. We use only the functional form (new value = function of old values) to allow other analyses such as constant propagation.

Each CFG edge has a precondition that is a Presburger formula that must be true when this edge is executed. Thus, the precondition is a *necessary* condition for

executing the edge, but perhaps not a *sufficient* one. For example, consider a step that decrements register *B*, jumps to another step if the result is not zero, and otherwise continues to the next step in address order. The edge to the next step has the precondition $B = 0$ while the edge for the jump has the precondition $B \neq 0$. An unknown precondition is represented by the constant *true*.

Since our aim is only to find loop bounds (not, for example, numerical errors such as division by zero), we only model those instructions and storage-cells that are likely to be used for loop counters. For floating-point instructions we only model the side-effects on the integer computation. On the ADSP-21020 for example, where any 32-bit general register can be used for integer computation or floating-point computation, the result of a floating-point computation is considered to be unknown.

We generally assume that the computation does not overflow or underflow. For the 8-bit Intel-8051 we provide a command-line option to negate this assumption. For example, if register *A* in the above example is 8 bits wide, unsigned arithmetic is used and overflow is considered, the effect of incrementing *A* would be encoded as *A'* = *if* *A* = 255 *then* 0 *else* *A* + 1. Unfortunately this creates many conditional assignments which slows down the Presburger solver markedly.

3.5 Program model

The program model consists of all the subprograms under analysis, starting from the "root" subprograms named by the user and including all their callees. We will use the term "call" to mean a specific step, in the CFG of a caller subprogram, that represents a call to a specific callee subprogram.

Each call is associated with a parameter-passing map between the cells in the caller and the cells in the callee. This map is used to propagate bounds on parameter values from the caller to the callee, perhaps for several call levels, in the hope that these parameters define loop bounds. We do not track the other data-flow direction, from callee to caller. The value of any cell that is modified in the callee is considered unknown after the return.

4 Analysis phases and methods

4.1 Overview

The analysis phases in Bound-T are, in order:

1. Reading the target program.
2. Instruction decoding and control-flow tracing.
3. Arithmetic analysis for dynamic branches.
4. Arithmetic analysis for dynamic data accesses.
5. Loop bounding analysis.
6. ILP analysis to find the worst-case path.

After reading in the binary program and its symbol tables, Bound-T traces the control-flow starting at the root subprogram entry-points. Each instruction is decoded, entered in the CFG as a step (or many steps, or part of a step), and the possible successors (new step addresses) are found and decoded in turn. When a call instruction is found, the callee is added to the set of subprograms to be analysed. This phase terminates when all paths end with a return instruction (a step with no successors) or a dynamic branch (successors so far unknown).

For subprograms with dynamic branches, arithmetic analysis is applied to try to resolve the target address of each branch. If this succeeds, the exploration of the control flow is resumed. There may be several iterations of flow-analysis and arithmetic analysis.

When the CFGs of all subprograms are complete, arithmetic analysis is applied to try to resolve dynamic data accesses. Unresolved accesses are left as such, and are considered to yield unknown values.

Loop bounding analysis tries to find cells that act as loop counters, to find bounds on the values of these counters, and thus bound the number of loop iterations.

When loop-bounds in a subprogram are known, the ILP analysis finds the worst-case path in the subprogram in the usual way as the solution of an ILP problem where the unknowns are the number of times each CFG node or edge is executed, the constraints are derived from the CFG and the loop-bounds, and the objective is to maximise the execution time.

The following sections explain the arithmetic analysis and the loop bounding analysis in more detail. The other phases use conventional methods.

4.2 Arithmetic analysis

In the arithmetic analysis of a subprogram, the arithmetic effects of several steps in the CFG are joined to give the overall effect of some execution path. In mathematical terms, the effect of a step is a relation between the cell values before and after the step, called the *input-output relation*. The input-output relation for a sequence of steps (a path) is computed simply by joining (chaining) the relations of the steps. The preconditions on edges in the path are included as additional constraints in the chain. In an acyclic part of a CFG, a simple one-pass algorithm can compute the input-output relation between any pair of steps. When there are several paths between the steps, the "incoming" relations to a step where the paths join are combined by set union (disjunction).

Loops are handled by a bottom-up traversal. The body of an innermost loop is acyclic, so we can compute the input-output relation of the body. From this relation, we find the cells that must be loop-invariant (output value = input value). The entire loop is then fused into one step with an effect that approximates the effect of the loop as an input-output relation that keeps the loop-invariant cells

constant and does not constrain the other cells, leaving them with unknown values. (To be precise, the relation does include the constraints created by the loop-termination paths, from the loop-head to a loop-exit, assuming unknown values at the loop-head.) The next higher (containing) level of loops can then be analysed and fused in the same way.

In some target processors every instruction sets many flags. This creates many conditional assignments in the effect of the instruction, but most of these are "dead" because the flag is redefined by another instruction before it is used. Before the actual arithmetic analysis as described above, we do a live-variable analysis in the normal way, and include only the live assignments in the input-output relations.

The results of this arithmetic analysis are the input-output relations from the subprogram entry-point to each step in the subprogram. From these relations we compute bounds on the values of cells at specific steps, for example bounds on the addresses of dynamic branches and data accesses, or bounds on the actual parameters in calls to lower-level subprograms. Derived or asserted bounds on the parameters of this subprogram, or on local or global variables, can define or sharpen the computed bounds.

4.3 Loop bounding analysis

Each loop is analysed separately as follows, in top-down and flow order. Let the *repeat relation* be the input-output relation that represents repetition of the loop, in other words all paths from the loop-head through the loop-body back to the loop-head. All the non-invariant cells are candidates for loop counters. For each candidate, we compute bounds on the candidate's *initial value* before the loop, on the *repeat value* implied by the repeat relation, and on the change in the value implied by the repeat relation. Consider for example this loop:

```
j := 3;
loop
   j := j + 2;
   if ... then j := j + 3; end if;
   exit when j > 9;
end loop;
```

For the cell j, the initial value is strongly bounded to $j = 3$, the repeat value is bounded by $j \leq 9$, and the change is bounded to $2 \leq \Delta j \leq 5$. Together, these imply that j is an up-counter and that the loop-head can be re-entered from the loop-body at most *ceil* $(9 - 3 + 1) / 2) - 1 = 3$ times, for a total of 4 iterations of the loop.

In this way we can discover up-counters and down-counters. By computing the complement of the repeat relation we can discover counters that terminate the loop on equality ("exit when $j = 9$"), but only if Δj is exactly 1.

5 ADSP-21020 program sequencing

Digital Signal Processors often have special support for loops, to speed up vector computation and digital filtering. The ADSP-21020 has an instruction of the form "DO address UNTIL condition" which sets the processor into 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 to the instruction after the DO, depending on the value of the condition flag two cycles earlier (due to pipe-line lag). Such loops can be nested to six levels and can interact in interesting ways with the delayed branch instructions.

For the ADSP-21020, we model in the step-address type the entire three-stage instruction pipe-line (fetch, decode, execute) and the whole six-level loop-nest. This has the interesting result that a single instruction can appear as several steps in a CFG. For example, an instruction that follows a conditional delayed branch is decoded twice and represented as two steps, because the step-address for the branch instruction has two successor step-addresses, one that represents the case where the branch is taken, and the other the case where it is not taken. These step-addresses differ in the "fetch" stage of the pipe-line model. The delayed branch instruction is converted into an immediate branch in the CFG.

For another example, consider a block of instructions that can be entered either through a DO UNTIL instruction, or directly without a DO UNTIL. This whole block is then decoded twice, once in a context with an active loop-level for this DO UNTIL, and once without.

6 SPARC IU/FPU concurrency

In the past year and with ESA support, we completed targeting Bound-T to the SPARC V7, including analysis of the register-file overflow and underflow traps and of the concurrency between the Integer Unit (IU) and the Floating Point Unit (FPU). In this paper, we focus on the IU/FPU concurrency.

6.1 The problem

In the SPARC V7, the IU is responsible for fetching all instructions and for executing integer instructions. Each floating-point (FP) instruction is forwarded to the FPU which executes the instruction while the IU fetches and executes new integer instructions. When a new FP instruction is found, but the FPU is still busy, the IU must wait until the FPU has finished the first FP instruction. The delay depends on the execution time of the first FP instruction, which increases the delay, and on the amount of IU computation between the two FP instructions, which decreases the delay. In the worst case, the delay

can be nearly 80 cycles. An integer instruction is typically executed in one or two cycles.

One difficult aspect of this problem is that the delay depends on the path taken by the IU between the two FP instructions; this can be a large number (up to 80) of integer instructions. The delay is not an interaction between two *consecutive* instructions. Another difficult aspect is that the integer-execution time appears with a negative sign in the expression for the delay, which means that we would need the *best*-case IU time for computing the *worst*-case delay.

6.2 Finding delayed paths

The first phase in the IU/FPU analysis is thus to find the (potentially) *delayed paths* between two FP instructions (or from an FP instruction to itself), where the intervening integer computation is too brief to ensure that the first FP instruction has finished before the second one should start. Delayed paths are found simply by depth-first CFG traversals starting at each FP instruction and propagating the maximum remaining FPU-busy time along all paths, until another FP instruction is found or the FPU is sure to have finished the first FP instruction.

We avoid the need for best-case IU times by a principle we call "hurry up and wait". If the IU executes the path between the two FP instructions *faster* than the worst-case bound, it will just have to wait *longer* for the FPU to finish the first FP instruction. In fact, the WCET of the first FP is a worst-case estimate for the whole path from the start of the first FP to the start of the second FP, including the FPU-busy delay before starting the second FP. Thus, we phrase the analysis in terms of the total time for each delayed path, not directly in terms of the delays.

6.3 Assigning delays to edges

Our objective is to minimize the pessimism, so that the FPU-busy delay is associated only or mainly with the paths that really incur delay in execution. The simplest approach would be to assign the worst-case delay time to each edge that enters the second FP instruction and that is part of a delayed path. This is pessimistic if this edge is part of the overall worst-case path and the worst-case path is not in fact delayed at this FP instruction. It could be less pessimistic to assign the delay to an edge that leaves the first FP instruction, if this edge is used only by the delayed path.

To avoid such pessimism, we use an ILP approach to distribute the worst-case blocking delay from *all* delayed paths onto *all* the edges in delayed paths so that the added pessimism is minimized (using a heuristic goal function, however). In this ILP problem, the unknowns are the additional delays to be assigned to the edges on delayed paths; the constraints are that the total execution time (including these additional delays) of every delayed path must be at least the WCET of the FP instruction at the start of the path; and the objective is to minimize an expression that estimates the pessimism.

6.4 Is it cheating?

When the FPU-busy delays have been distributed to the edges of delayed paths, the subprogram timing model — the CFG with times assigned to nodes and edges — is no longer a worst-case model, since the time assigned to the last edge and node of a delayed path is not necessarily an upper bound on the actual execution time of this edge and node. However, we can prove that the normal Bound-T WCET analysis (which assumes a worst-case model) still gives an upper bound on the execution of the whole subprogram, thanks to the constraints imposed on the distribution of the delays.

6.5 Inter-procedural aspects

To handle FPU-busy delays between FP operation pairs that cross subprograms, a bottom-up inter-procedural analysis assigns each subprogram a mimimum *margin* of pure IU running time on entry, and a maximum *legacy* of remaining FPU running time on return. These summary values are associated with the call steps in the analysis of the calling subprograms.

7 Marketing and commercialization

To commercialize Bound-T we have contacted a number of potential development partners, tool distributors and tool users, but the progress is very slow. WCET analysis is still poorly known, and it is often hard to make people understand what it does and how it differs from debugging, simulation and testing.

Partly because of the marketing delays, and partly because of staff shortages, no technical development is currently under way. SSF will itself use the SPARC V7 version in a major project that develops the on-board platform software for the ESA GOCE satellite (Gravity and Ocean Circulation Explorer). This will test the specific SPARC analysis methods described above, as well as the generic abilities of the tool.

We still hope to make Bound-T a commercially available tool, but the near-term plans are vague and depend on finding partners or users. We will gladly supply evaluation copies of Bound-T. The host platforms are Sun Solaris, Intel Linux, and Intel Windows (using CygWin and a command-line interface).

8 Future work

It is evident that the range of target processor should be increased and updated. Candidates for new targets include ARM, MIPS, AVR, and other microcontrollers.

We also have plans for several generic technical improvements. The derived loop-bounds could be used to sharpen the resolution of dynamic data accesses and dynamic branching, and the latter two should be iterated when needed, since some dynamic branching depends on dynamic data addressing (switch tables).

In the loop-bound analysis, it would be better to compute bounds on the difference between the initial value and the repeat value of a counter cell, instead of separate bounds on the two values. This would let us bound loops of the form "for I in N .. N + 10" even when the value of N cannot be bounded statically with any precision.

Nested loops where the bounds of the inner loop depend on the counter of the outer loop will currently yield pessimistic WCETs, because the worst-case bounds on the inner loop are assumed to hold for all executions of the outer loop. We don't have a clear idea how this analysis could be improved in the Presburger method.

Better aliasing analysis and optional levels of aliasing analysis will probably be necessary for large target programs. Finally, while the current command-line interface is quite workable, a GUI would be convenient for browsing the analysis results of large programs.

9 References

To be added in the final paper.

You Can't Control what you Can't Measure, OR Why it's Close to Impossible to Guarantee Real-time Software Performance on a CPU with on-chip cache

Nat Hillary
Manager of Technical Marketing
Applied Microsystems Corp.
nath@amc.com

Ken Madsen
Manager, Product Marketing
Wind River Systems, Inc.
ken.madsen@windriver.com

June 3, 2002.

1. Abstract

Steady increases in CPU core speeds continue to extend the range of applications for computer-based solutions, resulting in the creation of ever more responsive systems. At these higher core speeds, on-chip cache architectures are used to prevent the CPU from stalling when accessing relatively slow off-chip memory. In normal operation, most fetch-execute cycles occur internally, guaranteeing the execution of the maximum instructions per second. However, this also serves to hide the state of executing code from the user. Given the fact that it is not possible to directly monitor the execution of code within such a CPU when it is running at full speed, is it possible to guarantee and control the performance of Real-Time software on these cache-based CPU architectures?

This paper investigates this issue by first offering a definition of Real-Time software, together with a discussion on what must be done to prove that the system will meet its performance objectives in all circumstances. In addition, the range of software performance monitoring techniques that are currently available will be discussed, together with a summary of the pros and cons of each measurement technique. The conclusion of this paper is that it is close to impossible to make deterministic software performance measurements using traditional techniques on a CPU that heavily utilizes on-chip cache, so it is therefore almost impossible to guarantee the performance of Real-Time software based on these styles of microprocessor architectures unless new measurement techniques are utilized.

Real-Time software is distinguished from any other application by the fact that performance criteria are included in the specifications. This means that for Real-Time software to be verified as being correct, it has to be proven that the software will always meet its performance objectives. With cache based microprocessor architectures, software performance measurements are extremely difficult to do without causing serious perturbations, affecting measurement accuracy. This holds true across all possible measurement techniques. Accurate, hardware only performance measurements are generally not possible on architectures utilizing on-chip cache.

Software performance measurement techniques range from hardware assisted solutions, to software only ones. All techniques, however, rely on being able to get information about the current state of the executing code off-chip.

Pure hardware or hardware assisted solutions such as Logic Analyzers (LAs), In Circuit Emulators (ICEs), or dedicated pure software performance monitoring devices cannot determine what code is executing in the cache-based CPU core by monitoring external microprocessor signals. It is therefore necessary to force the activation of off-chip signals (such as an off-chip write or an assertion of a hardware signal) in order for the monitoring hardware to determine the state of the executing code in the CPU core. Due to the performance limitations on the external busses of cache-based CPUs, these external writes have the side effect of stalling the CPU, affecting the accuracy of any performance measurements based on them.

Software only solutions (such as instruction pointer sampling) do not require off-chip writes during measurement, but they introduce their own limitations. As they necessarily require extra software components to be executing on the CPU in addition to the application under test, they cause their own perturbations that affect software performance measurements. By placing extra demands on the CPU, these software measurements are limited in their accuracy. In addition, the introduction of additional code will affect cache flush and update intervals, significantly impacting the accuracy of any performance measurements.

Guaranteeing the performance of real-time software relies on being able to prove that the software will meet its performance objectives in all circumstances. As this paper suggests, obtaining accurate timing measurements is very difficult for systems utilizing CPUs with on-chip cache based architectures. Does this mean, then, that this type of CPU should not be used for real-time systems? The answer is no. This type of CPU is often ideally suited for maximum performance real-time systems and must often be used by system designers in order to build a system possessing top competitive performance characteristics. However, the full maximum real-time performance of these systems cannot be easily guaranteed with any single measurement approach.

This paper will show how the best approach to measuring real-time responsiveness for a system with a CPU containing on-chip cache is not a single measurement approach, but is in fact an approach based on the intelligent, clever, and often simultaneous, use of multiple measurement techniques, from pure hardware based techniques, to hardware assisted based techniques to pure software based techniques.

2. Real-Time Software

Because Real-Time software has performance criteria included in its specifications, it is essential that software execution performance be monitored at every step of the way while it is being created, from the writing of Interrupt Service Routines (interrupt service routines) to time-critical sections of application code. So what techniques may be used to measure software execution performance, and what are the implications of using them with a cache-based CPU?

Starting from board bring-up, the measurement technologies most commonly employed are:

- Logic Analyzers
- In Circuit Emulators
- Hardware-assisted software performance monitors
- Software-assisted software performance profilers

3. Logic analyzers

Typically used to monitor hardware signals, logic analyzers may also be used to make high-resolution measurements of software performance, normally for point-to-point type timing measurements.

All software performance measurements made with a logic analyzer require external CPU signal lines to be asserted when particular lines of code are reached. This results in very high-resolution timing for specific sections of executed code, and when measuring response time against external stimulus or events.

Measuring hardware/software interactions (such as interrupt latency times) is fairly straightforward; a single command is placed at the entry to the software routine in question that asserts a signal on an external CPU pin (e.g. a spare chip select or programmable I/O pin, which will not stall the CPU). The logic analyzer is then used to measure the interval between the external hardware event and the signal marking entry to the software routine being asserted. The fine-grained timing measurements obtained using this technique may also be used for monitoring critical sections of code. Modern Logic Analyzers (such as the TLA series from Tektronix) extend this technique, allowing specific networking signals (such as Ethernet packets or ATM cell contents) to be used as hardware trigger events.

By using instrumentation (e.g. adding statements at salient points in code), logic analyzers may also be used for monitoring the performance of a Real-Time application.

For some applications, using spare off-chip signals are restrictive, as a prohibitive number would be needed in order to correctly identify each unique point in code. Instead, external writes are required to ensure that enough unique instrumentation values are included for the measurements to be meaningful. However, the resolution of this type of solution is slightly less than the technique described above for point-to-point measurements.

Although Logic Analyzers provide a means of making deterministic A to B type measurements of code, they typically do not gather profiling data over a statistically long period. It is therefore necessary to use analytical techniques to ensure that the correct conditions are created so that particular measurements accurately reflect the worst-case execution time of a particular section of code.

Logic Analyzers are an excellent solution for making controllable, minimally intrusive measurements of critical sections of code, particularly those associated with external hardware events.

In some rare cases, inserting additional code into an application degrades the performance to a point where the system's Real-Time characteristics are not being met. In this case, 'black box' performance testing techniques are required, where measurements are made at points external to the CPU. E.g. the response time between a particular Ethernet packet arriving and the system responding might be measured using a Tektronix TLA Logic Analyzer.

4. In circuit emulators

Typically used in the early debug stages of target board bring-up, in circuit emulators may also be used for software performance measurements.

Traditionally, the Real-Time bus trace capability was the most significant feature of an ICE. For non cache-based CPUs, bus trace can be used to monitor the timing of higher-level application code, including those that need to respond to external hardware events.

Aside from the lack of profiling data, this would be the ideal solution for performance monitoring of true Real-Time software, but it requires an off-chip fetch-execute cycle to occur in order to monitor what's going on.

Modern cache-based CPUs tend not to have full ICE solutions available. Instead, most modern CPUs have emulators that use serial test access points such as JTAG.

Most JTAG emulation solutions do not have Trace measurements. Triggering timing measurements with a JTAG emulator requires the use of hardware or software breakpoints, which are intrusive. In addition, the serial JTAG bus is slow by comparison to processor speed and events are detected asynchronously to their occurrence. Any timing measurements made via this bus are going to be subject to inaccuracies; monitoring the execution speed of a 400 MHz CPU core by sending information through a significantly slower serialized communications bus is not an ideal solution.

Traditionally the ideal solution for making software performance measurements on the fly, contemporary ICE solutions rarely support the features required to make deterministic timing measurements of code.

5. Hardware-Assisted Software Performance Monitors

An extension of in circuit emulation technology, hardware assisted software performance monitors, such as the CodeTEST product from Applied Microsystems, are designed specifically to measure software execution performance.

This technology requires the combination of software instrumentation and hardware data collection, with time stamping. It may be used to monitor low-level code (such as interrupt service routines), application level code and also RTOS activity. In addition, time stamping may be triggered by external events, making the timing of hardware/software interactions (such as interrupt latencies) possible.

Although the instrumentation of code is automatic, this technique requires an off-chip write for each measurement point, producing the same inaccuracies as with the Logic Analyzers above. However, the instrumentation required for monitoring the worst-case execution time of critical sections of code may be limited to a single manually inserted statement. This technique may also be used to gather performance data over a significant period of time, with the automatic collation of minimum, maximum and average execution times. The overhead of a single off-chip write is minimal, and is easy to calculate, making the measurements that this technique provides highly accurate and deterministic.

This technology provides the best method for monitoring critical sections of code and for general code optimization, by providing application level profiling data that identifies where the system is spending its time, ensuring that optimization efforts are focused on the right areas.

The 'call-pair' data provided by this technology may also be used to improve software performance. 'Call-pairs' measurements identify highly inter-dependent functions that make good candidates for either inlining, fixing in cache, being located close to one another in the link map of the application.

This technique has been successfully used in the CodeTEST product for Performance, Coverage, and Memory analysis, in addition to Software Execution Trace.

6. Software-Assisted Software Performance Profilers

Worthy of mention because of their dominance in the desktop marketplace, software-assisted performance profilers use a variety of techniques for monitoring where an application is spending its time. If this technology is ever used during the development of a Real-Time system, it is used to aide optimization efforts, and not to measure any of the Real-Time characteristics of the code.

Typically consisting of an in-target data collection agent and either code instrumentation or stack/IP sampling, the potential of this technology is intriguing for two reasons. First, these techniques do not require any off-chip accesses in order to make their measurements. Secondly, solutions based on these techniques tend to be extremely easy to use.

On the other hand, these techniques rely on a target based data collection agent, which is intrusive. Any techniques based on stack/IP sampling are also prone to aliasing, and in require higher levels of intrusion to improve their accuracy.

7. What Level of measurement accuracy is required?

For Real-Time systems, 'Real-Time' does not necessarily equate to 'real-fast'. The environment in which a system must operate dictates the performance criteria of Real-Time software. A pacemaker, for instance, must respond to specific physiological events within a specific time period before permanent damage to the heart ensues (response times in the 100's of mS). Meanwhile, a commercial flight control system must process and respond to thousands of inputs a second, from pilot commands to air data (response times in the mS).

With modern CPUs capable of processing in excess of 2 billion instructions per second, is it really necessary to measure software performance on a per instruction basis?

The simple answer is no, provided that:

- Worst-case response/execution times of a system are monitored, verified and managed
- Enough information is to hand during software creation to ensure that the system performance objectives can be met.

From this, then, the question then arises whether this is achievable with CPUs utilizing on-chip cache.

For extremely high accuracy software performance measurements of worst-case execution time (e.g. nS accuracy), Logic Analyzers may be used. Alternatively, if uS accuracy of software performance is required, then hardware assisted software performance monitoring technologies show the most promise. The only question is whether the performance impact of the off-chip writes that these technologies require is prohibitive, or not. This is worth a more detailed consideration.

When measuring the worst-case execution time of a critical section of code (e.g. the main loop in a control function) using this technology, a single write statement is required. Timing is started when the write occurs the first time, and then the interval between each occurrence is timed. But what overhead does this introduce?

Consider a typical environment where a target system is using a 100 MHz external CPU bus that requires 3 clock cycles to complete a write operation. In this instance, the delay imposed by each write operation would be a deterministic 30 nS.

The impact of a 30nS delay per cycle in the time critical code of a Real-Time system is negligible. The impact of being able to deterministically measure the worst-case execution time of the software under measurement with uS accuracy, however, is not. This lends great credence to the power of hardware assisted software performance monitoring technologies, especially when these technologies may be used to gather timing information on the critical sections of code over a significant period of time, ensuring the true worst-case execution time is understood.

8. Conclusion

It is an age-old dilemma in science; how can you measure something without affecting it? When it comes to measuring the performance of Real-Time software, the simple answer to this is – you can't. Add a CPU that utilizes on-chip cache, and the situation only gets worse. It is imperative, therefore, that the right performance measurement technique be used for the software being created. If the Real-Time nature of the software under development requires a timing accuracy in the nS range, then a Logic Analyzer must be used for software performance measurements. It must be understood, however, that data can only be gathered over a limited measurement period. Therefore, careful consideration must be made in the creation of the stimulus or circumstances to make sure that the worst case scenarios are represented for measurement and analysis.

Traditionally, Logic Analyzers required intimate knowledge of memory implementations on the target hardware, thus they provided very little functionality for software engineers. However, new products such as LA Trace from Wind River Systems abstracts the bus implementation from the user making it easy for software engineers to configure the circuitry of a Logic Analyzer to make complex timing measurements. Furthermore, Wind River's LA Trace is able to leverage RTOS knowledge to present acquired information relative to RTOS threads and events.

On the other hand, if you want information in the uS range, use the type of hardware assisted software performance monitoring technology available with the CodeTEST product from Applied Microsystems. This not only provides accurate one-shot timing information, but it also gathers performance information over an indefinite period of time, ensuring that the worst-case execution time of the software being measured is encountered. In addition, the same technology provides function profiling data that greatly enhances optimization efforts, and call-pair information that enables immediate performance improvements through in-lining or prudent link-map ordering.

Real-Time bus trace data from in circuit emulators have traditionally the fall back solution for Real-Time software performance measurements. Most modern CPUs utilizing on-chip cache, however, only have serial JTAG emulation solutions without Real-Time bus trace

capabilities. Emulators do not, therefore, provide the performance information that they once did.

Software only profiling solutions, popular in the desktop market, are too intrusive and/or inaccurate to make accurate worst-case execution time measurements for Real-Time systems. However, they do provide the profiling information that may be used to yield significant performance improvements during code optimization.

As with all measurements in science, it is impossible to measure the worst-case execution time of Real-Time software without affecting the system. Nevertheless, technologies are available that are appropriate for the required level of accuracy, ensuring that the Real-Time nature of software executing on a CPU utilizing on-chip cache can be controlled.

The European Space Agency's involvement and interest is WCET and scheduling analysis

Extended Abstract

Morten Rytter Nielsen, ESA (morten.nielsen@esa.int)
Eric Conquet, ESA (eric.conquet@esa.int)
Jean-Loup Terraillon (jean-loup.terraillon@esa.int)

Abstract

We consider the use of scheduling analysis as not being a standalone exercise but a system-level activity, congruent with the conscious decision for a 'correctness by construction' development model. We describe how ESA have incorporated the ideas of scheduling analysis into our required standard practices; how we have ensured that the enabling technology is available; and where we see the future of WCET technology and scheduling analysis.

The development approach

The traditional development model, which is used in software space projects under the responsibility of ESA, follows the classical waterfall V-model [ESA-PSS-05]. In this development model the User Requirements are ESA's requirements towards industry and the Software Requirements (or Technical Requirements) are industry's refinement of these. These Software Requirements are then followed by Architectural Design, Detailed Design and coding. On the ascending part of the V-model Unit Tests (verifies Detailed Design), Integration Tests (verifies Architectural Design), System Tests (verifies Software Requirements Definition) and Acceptance Tests (verifies User Requirements) are performed. The testing effort is usually 50-60 percent of the total development effort. Each phase of the development model is finalized with reviews and acceptance together with associated payments.

Historic Space Systems

The V-model has proven its value through many years and projects. Traditionally onboard software-systems have been quite simple and with well separated functional blocks. The utilized software technology centered on fixed cyclic schedulers and dedicated proprietary kernels and very often the I/O mechanism was polling or well characterized interrupts. The required method in the ESA standards for controlling the performance behavior was limited to requirements for CPU utilization at the different stages of development (projects typically used 50% at Architectural Design, 60% at Detailed Design and 70% on final acceptance of the software code). Real-time requirements in the form of reactivity/responsiveness and jitter where either non-existent or at best occasional. The CPU utilization was typically acquired by estimation and later by measurement performed on the final code.

Current Space Systems

The new generation of onboard space systems is significantly richer in functionality and complexity, with much more interaction between functional blocks than traditional onboard systems. Among other reasons, this trend originates from:

- Added throughput (dedicated services)
- More intelligent Autonomy and Failure, Detection, Isolation and Recovery (FDIR) functionality
- Intelligent instruments that sporadically interrupt the main computer
- Added capability of the onboard system in general

Many real-time requirements are now part of the requirement baseline to ensure reactivity and enable different units of the satellite to be developed to lesser tolerances.

Several new problems have surfaced in the new generation of onboard systems [ESA STR-260]. Many of these problems occur in the real-time behavior area. Since CPU utilization is not a sufficient way to ensure real-time behavior, the development approach has to be adapted. ESA have thus sponsored and funded a number of initiatives and supported (and still supports) the introduction of scheduling analysis in the ways outlined in the following paragraphs. For us it is clear that the use of scheduling analysis have major repercussions on the implementing technology as well as on the process standards and associated development approach. This altogether raises a clear demand for better tools support, not limited to the extraction of WCET and the scheduling analysis but also extending to the specification of the real-time attributes and properties of the system.

Standards

The new European generation of space standards, the ECSS standards, allows more flexible development approaches to be used (e.g. spiral models and rapid prototyping) [ECSS-E40B-July2000 and ECSS-E40B-Feb2002]. However, they also require that the used computational model of the system be identified. This explicitly includes the component types (e.g. active-periodic, active-sporadic, protected, passive, actors and process), the assumed scheduling type and model (e.g. fixed priority or dynamic priority) and the accompanying analytical model under which the model is executed (e.g. Rate-Monotonic Scheduling or Deadline-Monotonic Scheduling).

This evolution shows that the previously informally used CPU utilization is now being replaced by much more stringent requirements on the chosen architecture and the rationale behind this choice.

Projects may decide to waive requirements in the standards if this implies too much effort. Thus the enabling technology is very important to lower the entrance to applying scheduling analysis.

Enabling technology

Specification and Design level

In order to be able to really harvest the benefits of the scheduling technology early in the development process, ESA saw the need to accommodate the computational model already at design level. The result of this effort is the HOOD derived HRT-HOOD method [HRT-HOOD]. Currently, TNI (France) and Intecs Sistemi (Italy) have commercial tools supporting this specification and design method.

Implementation technology

ESA have supported the Ada Ravenscar definition from a user perspective. Furthermore we have funded the development of the GNAT/ORK kernel and compilation system and the port of Aonix Raven to the space processor ERC32. Also CNS have an Ada Ravenscar system for the ERC32. Ravenscar compilation systems are now used for Beagle2 and GOCE.

WCET extraction
In some projects the extraction of the WCET profile have been done by hand. However, for scheduling analysis to be used widely and systematically in the space domain, we believe that tools supporting this process are needed. Various ways of acquiring the WCET have been tried, including:

- Instrumentation: Logic analyser (Tektronik) and embedded instrument code (Aonix and VxWorks/Tornado) plus user developed instrument code
- Source level analysis with the support of the compiler: a prototype based on the Adaworld compiler have been developed by Aonix
- Static Analysis on image code: Bound-T from SSF (Finland) have been developed for both the DSP 21020 and the ERC32

Scheduling analysis
Tools to help apply different scheduling analysis techniques have been developed and are now available from Spacebell (Belgium). These tools assist in margin analysis and enables persons less fluent in the logic behind the analysis to interpret and evaluate the results.

Test cases
A standardization of core onboard services has taken place in the form of the Packet Utilization Standard [ESA-PSS-05]. OBOSS [OBOSS] is a reference implementation of selected services, which have been used as a guinea pig for scheduling analysis and the Ada Ravenscar profile. Furthermore, the development approach using scheduling analysis and thereby moving the verification of real-time properties from the typical integration testing phase to the specification and design phase have been applied with great success on the European Robotic Arm (ERA) which is a safety critical module to be used on the International Space Station.

Future of Space Systems

The new draft ECSS standards for onboard space engineering require that scheduling analysis must be performed. Several proposals for new onboard systems are base-lining Ada Ravenscar as the implementation technology and the awareness of scheduling analysis is increasing. Together with standards that require a strong development baseline and a consolidation of the tools assisting in the scheduling analysis in all relevant phases of the development process, the entry barrier for the application of this development approach will be continuously lowered.

ESA continues to fund and promote the development of the enabling technology and the support for the development approach referenced in this paper. The near future evolution is the new space processor LEON, which like the current ERC32 has a Sparc instruction set. The transition to LEON, which has cache, raises new challenges that will require 'expert support' in addressing.

The movement and activities described in this paper has been triggered by problems encountered in space projects using the current development approach. These activities focus on a single computation platform with embedded software. As space onboard systems are moving from synchronous to asynchronous behavior, the need to extend the scheduling analysis to system level is surfacing. ESA is participating in organizations supporting the AADL (Avionic Architecture Description Language) standard. The aim of this work is to define a common language for the design and verification of complex avionic systems. We expect from such a standardization effort the emergence of an open framework that can incorporate various design languages and verification tools able to trap performance and behavioral issues in early design phases.

Conclusion

We have in this extended abstract explained the context and the support of the WCET and scheduling analysis in ESA and the problems that we have encountered which let to this. In the full paper we will include experiences of the different areas outlined above and expand on the future as ESA sees it. This will include specific activities started or foreseen to be started in the area of distributed scheduling analysis.

References:

[ESA-PSS-05] ESA PSS-05-0 Issue 2, February 1991: ESA Software Engineering Standards

[ESA-PSS-07-101] ESA PSS-07-101 issue 1, May 1994: Packet Utilisation Standard

[ECSS-E40B-July2000] ECSS-E-40B Draft 1, 28 July 2000: Space Engineering. Software

[ECSS-E40B-Feb2002] ECSS-E40B Draft 1, 15 February 2002: Space Engineering. Software

[HRT-HOOD] Burns, A. and Wellings, A.: HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems, Elsevier, 1995.

[OBOSS] OBOSS home page: http://spd-web.terma.com/Projects/OBOSS/Home_Page

[ESA STR-260] Vardanega, V.: Development of On-Board Embedded Real-Time Systems, ESA STR-260 October 1999

Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems

Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
e-mail: puaut@irisa.fr

Abstract

Cache memories have been extensively used to bridge the gap between high speed processors and relatively slow main memories. However, they are source of predictability problems and need special attention to be used in hard real-time systems. A lot of progress has been achieved in the last 10 years to model caches, in order to determine safe and precise bounds on (i) tasks WCETs in the presence of caches ; (ii) cache-related preemption delays. An alternative approach to cope with caches in real-time systems is to statically lock their contents so as to make memory access times and cache-related preemption times entirely predictable. This paper attempts to evaluate qualitatively and quantitatively the pros and cons of both classes of methods.

1 Caches and real-time systems

Extensive studies have been performed on schedulability analysis to guarantee timing constraints in hard real-time systems. Schedulability analysis methods assume that task worst-case execution times (WCETs) are known. While many schedulability analysis methods consider that the cost of task preemption is zero to simplify the analysis, some methods account for task preemption costs (e.g. manipulation of task queues, cache-related preemption delays).

Caches are small and fast buffer memories used to speed up the memory accesses. They contain memory blocks that are likely to be accessed by the CPU in the near future. Although the caches are a very effective means of speeding up the memory accesses in the average case, they are a source of predictability problems, due to intra-task and inter-task interferences. *Intra-task* interferences occur when a task overrides its own blocks in the cache due to conflicts, while *inter-task* interferences arise in multitasking systems due to preemptions. The inter-task interferences imply a so-called *cache-related preemption delay* to reload the cache after a task is preempted.

Caches raise predictability issues in hard real-time systems because they are designed to speed up the system *average case* performance rather than the system *worst case* performance which is of prime importance in hard real-time systems. As a consequence, the designers of hard real-time systems may choose not to use cache memories at all, or may choose to use on-chip static RAM - scratchpad memories – instead of caches [2]. The simple approach consisting in assuming that every access to memory results in a cache miss, causes the tasks WCETs to be largely overestimated, which may cause the schedulability analysis to fail while the system may actually be feasible. The main issue is then to estimate tasks WCETs and cache-related preemption delays in a safe but not overly pessimistic manner.

Two classes of approaches, described hereafter, can be used to deal with caches in real-time systems.

Cache analysis methods. A first class of approaches to deal with caches in hard real-time systems is to use them without any restriction, and resort to *static analysis* techniques to predict their worst-case impact on the system schedulability.

At the intra-task level, static WCET analysis techniques have been extended to predict the impact of cacheing on the WCETs of the tasks. They achieve a classification of the memory accesses regarding the instruction or data caches (e.g. hit when it can be proved that the access always results in a cache hit, miss otherwise). Techniques to predict the worst-case task behavior regarding the instruction cache can use data-flow analysis on each task control flow graph [12], abstract interpretation [1], integer linear programming techniques [10], or symbolic execution [11].

At the inter-task level, work has been undertaken to obtain safe and precise estimates of the cache-related preemption delay [9]. In this work, at every possible preemption point, the blocks that will be used by each task after that point are determined by static analysis, thus avoiding considering that the whole memory accessed by the task has to be reloaded in the cache after a preemption.

Cache partitioning and cache locking. A second class of approaches to deal with caches in real-time systems is to use them in a restricted or customized manner, so as to adapt them to the needs of real-time systems and schedulability analysis.

Cache partitioning techniques [8, 5, 14] assign reserved portions of the cache (partitions) to certain tasks in order to guarantee that their most recently used code or data will remain in the cache while the processor executes other tasks. The dynamic behavior of the cache is kept within partitions. These techniques eliminate the inter-task interferences, but need extra-support to tackle intra-task interference (e.g. static cache analysis) and reduce the amount of cache memory available for each task.

Another way to deal with caches in real-time systems is to use *cache locking techniques*, which load the cache contents with some values and lock it in order to ensure that the contents will remain unchanged [6]. This ability to lock cache contents is available on several commercial processors. The cache contents can be loaded and locked at system start for the whole system lifetime (*static cache locking*), or changed during the system execution, like for instance when a task is preempted by another one (*dynamic cache locking*). The key property of static cache locking is that the time required to access the memory is *predictable*.

Schedulability analysis for systems with caches. Some schedulability analysis methods (Rate Monotonic Analysis – RMA, Response Time Analysis – RTA) have been extended to cope with cache-related preemption delays in [3] and [4] respectively. They add the parameter γ_i, the cache-related preemption delay, to the formulas in charge of verifying the system feasibility (e.g. $\sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i} \leq n(2^{\frac{1}{n}}-1)$ in RMA for n tasks of period P_i and WCET C_i).

2 Cache analysis vs static cache locking

In the following, we give some elements that allow to choose between using statically locked caches or using the dynamic features of the caches, which imposes to use cache analysis techniques to bound accurately tasks WCETs and cache-related preemption delays. A static cache locking strategy with a frozen cache contents for *all tasks* is considered hereafter.

2.1 Qualitative comparison

Static cache locking is attractive from several point of views. First of all, it improves the system performance compared to a system that does not use caches, with respect to both average and worst-case system performance.

In addition, with static cache locking, the time required to perform a memory access is *predictable* (it is either a hit or a miss depending on whether the value is locked in the cache or not). While WCET analysis is still required, it alleviates the need for using complex cache analysis techniques for computing WCETs and cache-related preemption delays, and results in more simple WCET analysis tools. In particular, it eliminates the issue of integrating cache analysis techniques with the analysis techniques for the other architectural features (pipelines, branch prediction, etc).

Static cache locking can also be used when no cache analysis method can apply, due for instance to non-deterministic or poorly documented cache replacement strategies (e.g. pseudo-random replacement policies).

Another important benefit of static cache locking is that the technique addresses both intra-task and inter-task interferences, which is unique among the cache management techniques presented above. Concerning inter-task interferences, since in static cache locking schemes the cache blocks are statically partitioned among tasks, the cache-related preemption delay is null, or is constant and equal to the time required to reload the processor prefetch buffer if the processor is equipped with such a architectural feature. This low cache-related preemption delay is particularly important for large caches (see section 2.2).

However, statically locking the contents of instruction caches reduces the amount of cache memory available for each task. In addition, it raises the issue of selecting the cache contents. As we are interested in hard real-time systems, the main objective of the cache selection algorithm is to improve the worst-case system behavior according to some of the metrics used by schedulability analysis methods, such as CPU utilization or interferences between tasks. The main issue is then to avoid performing an exhaustive search of all possible cache contents, which would require an untractable computation cost. For instance, if every cache block can contain 4 program lines, checking the feasibility of the system with all possible cache contents would require 4^B feasibility tests, with B the number of cache blocks. This complexity led [6] to select a genetic algorithm for the selection of the cache contents and [13] to base the selection of cache contents on actual traces of the system execution.

Another potential benefit of static cache locking, although not proved yet by any study, is that it can easily apply to data caches, unified caches of multi-level caches.

2.2 Quantitative comparison

Since the primary focus in hard real-time systems is to prove that all deadlines are met, the key performance metric to be considered when comparing cache management schemes is the *worst-case* performance of the system. In this section, we compare the worst-case performance of a small task set made of periodic tasks (table 1 shows the task

Task name	Description	Code size (Bytes)	WCET-miss	Period
qurt	Computation of roots of quadradic equations	1824	21474	59697
minver	Matrix inversion	4320	36701	70098
jfdctint	JPEG integer implementation of the forward DCT	3440	29324	127559
fft1	FFT (Fast fourier transform) Cooly-Turkey algorithm	3620	115152	601093

Table 1. Task set characteristics

Asso \ Size		512B	1KB	2KB	4KB	8KB	16KB
1	Locking	+0.779	+0.665	+0.576	+0.517	+0.517	+0.517
	Analysis	+0.547	+0.413	+0.382	+0.388	+0.388	+0.388
2	Locking	+0.775	+0.658	+0.518	+0.459	+0.420	+0.420
	Analysis	+0.638	+0.439	+0.382	+0.388	+0.388	+0.388
4	Locking	+0.779	+0.623	+0.485	+0.418	+0.375	+0.368
	Analysis	+0.788	+0.609	+0.414	+0.389	+0.388	+0.388
8	Locking	+0.775	+0.622	+0.491	+0.415	+0.368	+0.368
	Analysis	-1.039	+0.771	+0.578	+0.421	+0.389	+0.388
16	Locking	+0.777	+0.602	+0.485	+0.414	+0.368	+0.368
	Analysis	-1.011	-1.035	+0.744	+0.585	+0.421	+0.389
32	Locking	+0.777	+0.602	+0.484	+0.412	+0.368	+0.368
	Analysis	-1.076	-1.007	-1.004	+0.750	+0.585	+0.421

Table 2. Compared worst-case performance of static cache locking and cache analysis

set characteristics[1]) using a state of the art *cache analysis technique* with its worst-case performance obtained using *static cache locking*.

The static cache locking algorithm implemented [13] selects the contents of the statically locked cache according to the knowledge of the tasks memory accesses, obtained using simulation. It locks the mostly used program lines of the tasks in the cache, in order to minimize the worst-case CPU utilization ($\sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i}$, with C_i, P_i and γ_i denoting respectively the WCET, period and cache-related preemption delay of task i)

We compare the worst-case performance of this task set with the one obtained through the use of a state of the art cache analysis technique based on F. Mueller's work on static cache simulation (see [12, 7] for details). The Heptane tree-based WCET analysis tool [7] has been used to compute WCETs. No attempt is made here to bound the cache-related preemption delay γ_i precisely (it is assumed that all program lines of a given task have to be reloaded after a preemption, with a maximum of N reloads where N is the number of cache lines).

The worst-case system performance of the task set is given in Table 2. Each cell indicates whether the task set is feasible or not according to CRTA [4] (Response Time Analysis enhanced with the knowledge of cache-related preemption delays γ_i , that are null for static cache locking and considered maximum for cache analysis). A '+' sign means that the task set if feasible, whereas a '-' sign means that it is not. The CPU utilization of the task set is also given in each cell. These two pieces of information are given for different cache sizes (Bytes), degrees of associativity, and this with and without static cache locking.

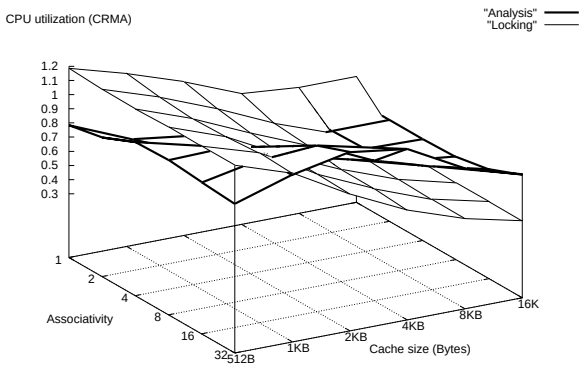


Figure 1. Worst-case CPU utilization

Figure 1 depicts the CPU utilization obtained on the task set from the contents of table 2. It compares the CPU utilization obtained when using cache locking and static cache

[1]In the table, delayes are expressed in number of processor cycles for a MIPS processor with a simplified timing model. WCET-miss denotes the WCET of the task assuming that all instructions cause a cache miss.

analysis. It can be noted that for a given degree of associativity, the performance of both static cache locking and static cache analysis increases with the cache size, because of the decrease of the number of conflicts for cache blocks. However, the performance increase of static cache locking is higher than the one of static cache analysis when the cache size increases. This is because the cache-related preemption delay increases linearly with the cache size for the static cache analysis method, whereas it stays constant for the static cache locking method.

For a given cache size, the performance of static cache locking scales better than the one of static cache analysis with an increasing degree of associativity W. Indeed, static cache locking takes benefit of the increasing degree of associativity to eliminate both intra-task and inter-task interference, which explains that the CPU utilization increases with W. In contrast, the static cache analysis method we have used does not scale well with W.

3 Open issues

The key benefits of static cache locking is to make the time required to perform memory accesses predictable, and to be a unified technique to take into account both intra-task and inter-task conflicts for cache blocks. This class of techniques alleviates the need for using complex static analysis techniques for computing WCETs and cache-related preemption delays. In addition, it can be applied in situations where static cache analysis cannot be used at all (*e.g.* when the instruction cache has a non deterministic or non documented cache replacement policy). While algorithms already exist for selecting the contents of statically locked caches [6, 13], we think that further work is required:

- to study their performance on larger real (non synthetic) benchmarks, in particular in task sets whose size is much larger than the cache size. For large programs, a possible direction is to explore more dynamic cache locking strategies (for instance, to select different contents of the locked cache changed at statically-defined points in order to cope with the tasks dynamic behavior while staying predictable)
- to study the impact of statically locked caches on the system average case performance
- to study the applicability of static cache locking techniques to data/unified/multi-level caches
- to address implementation issues on actual embedded processors
- to compare the use of statically locked caches with the use of on-chip static RAMs (benefits wrt predictability, issues to be addressed)

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[2] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Atlanta, GA, USA, Nov. 2001.

[3] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1994.

[4] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 204–212. IEEE Computer Society Press, June 1996.

[5] J. V. Busquets-Mataix and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proc. of the 9th Euromicro Workshop of Real-Time Systems*, pages 56–63, Toledo, Spain, June 1997.

[6] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask premptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, London, UK, Dec. 2001.

[7] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[8] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS89)*, pages 229–237, Santa Monica, California, USA, Dec. 1989.

[9] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), June 1998.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, IEEE Computer Society Press, Dec. 1996.

[11] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, Nov. 1999.

[12] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[13] I. Puaut. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. Submitted to publication - available on demand, May 2002.

[14] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, Aug. 1993.

A Framework to Model Branch Prediction for WCET Analysis

Tulika Mitra
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543
tulika@comp.nus.edu.sg

Abhik Roychoudhury
Department of Computer Science
School of Computing
National University of Singapore
Singapore 117543
abhik@comp.nus.edu.sg

In this paper, we present a framework to model branch prediction for Worst Case Execution Time (WCET) analysis. Our micro-architectural modeling is completely generic, and parameterizable w.r.t. the currently used branch prediction schemes. It automatically derives linear constraints on the total misprediction count from the control flow graph of the program. These constraints can be solved by any integer linear programming (ILP) solver to estimate the WCET.

Current generation processors perform control flow speculation through branch prediction, which predicts the outcome of branch instructions. If the prediction is correct, then execution proceeds without any interruption. For incorrect prediction, the speculatively executed instructions are undone, incurring a branch misprediction penalty between 3-19 clock cycles. If branch prediction is not modeled, all the branches in the program must be conservatively assumed to be mispredicted for finding the WCET. This pessimism results in as much as $60 - 70\%$ overestimation for some of the benchmarks in this paper, even assuming a 3 clock cycle branch misprediction penalty.

A classification of branch prediction schemes appears in Figure 1. Branch prediction can be *static* or *dynamic*. Static schemes associate a fixed prediction to each branch instruction via compile time analysis. Almost all modern processors, however, predict the branch outcome dynamically based on past execution history. Dynamic schemes are more accurate than static schemes, and in this work we study only dynamic branch prediction. The first dynamic technique proposed is called *local branch prediction* [4], where each branch is predicted based on its last few outcomes. This scheme uses a 2^n-entry *branch prediction table* to store the past branch outcomes, which is indexed by the n lower order bits of the branch address. In the simplest case, each prediction table entry is 1-bit and stores the last outcome of the branch mapped to that entry. When a branch is encountered, the corresponding table entry is looked up and used as the prediction. When a branch is resolved, the corresponding table entry is updated with the outcome. A more accurate version of local scheme uses k-bit counter per table entry.

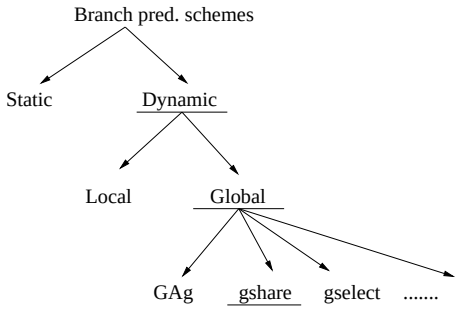


Figure 1: Classification of Branch Prediction Schemes. At each level, the more widely used category is underlined.

Most modern processors however use *global* branch prediction schemes [4] (also called correlation based schemes), which are more accurate. Examples of processors using global branch prediction include Intel Pentium Pro, AMD, Alpha as well as embedded processors IBM PowerPC 440GP and SB-1 MIPS 64. In these schemes, the prediction of the outcome of a branch I not only depends on I's recent outcomes, but also on the outcome of the other recently executed branches. Global schemes can exploit the fact that behavior of neighboring branches in a program are often correlated. Global schemes uses a single shift register, called *branch history register (BHR)* to record the outcomes of n most recent branches. As in local schemes, there is a global *branch prediction table* in which the predictions are stored. The various global schemes differ from each other (and from local schemes) in the way the prediction table is looked up when a branch is encountered.

Little work has been done to study the effects of branch prediction on WCET. Effects of static branch prediction have been investigated in [1, 3]. However, most current day processors (Intel Pentium, AMD, Alpha, SUN SPARC) implement dynamic branch prediction schemes, which are more difficult to model. To the best of our knowledge, [2] is the only other work on timing estimation under dynamic branch prediction. However, their technique only models the effects of local prediction schemes.

The starting point of our analysis is the control flow graph (CFG) of the program. Let v_i denote the number of times block i is executed, and let $e_{i,j}$ denote the number of times control flows through the edge $i \rightarrow j$. As inflow equals outflow, $v_i = \sum_{j \rightarrow i} e_{j,i} = \sum_{i \rightarrow j} e_{i,j}$. We provide bounds on the maximum number of iterations for loops and maximum depth of recursive invocations for recursive procedures. These bounds can be user provided, or can be computed off-line for certain programs.

Let $cost_i$ be the execution time of basic block i assuming perfect branch prediction. Given the program, $cost_i$ is a fixed constant for each i. Then, the total execution time of the program is $\sum_i (cost_i * v_i + penalty * m_i)$ where $penalty$ is a constant denoting the penalty for a single branch misprediction; m_i is the number of times the branch in block i is mispredicted. By maximizing this objective function we can get WCET.

Modeling Prediction Schemes To determine the prediction of a block i, we first compute the index into the prediction table. We define v_i^π and m_i^π: the execution count and the misprediction count of block i when branch in i is executed with index = π. By definition:

$$m_i^\pi \leq v_i^\pi \qquad m_i = \sum_\pi m_i^\pi \qquad v_i = \sum_\pi v_i^\pi$$

The prediction schemes differ from each other primarily in how they index into the prediction table. To predict a branch I, the index computed can be a function of: (a) the past execution trace (history) and (b) address of the branch instruction I. In the *GAg* scheme, the index computed depends solely on the history and not on the branch instruction address. Other global prediction schemes (*gshare*, *gselect*) use both history and branch address, while local schemes use only the branch address.

Our modeling is independent of the definition of the prediction table index π. Hence it can apply to any branch prediction scheme that uses a single prediction table. To model the effect of different branch prediction schemes, we only alter the meaning of π, and show how π is updated with the control flow.

In the case of *GAg*, this index is the outcome of last k branches before block i is executed. These k outcomes are recorded in the Branch History Register (BHR). To model the change in history due to control flow, we use the left shift operator ; thus $left(\pi, 0)$ shifts pattern π to the left by one position and puts 0 as the rightmost bit. We define:

Definition 1 *Let $i \rightarrow j$ be an edge in the control flow graph and let π be the BHR content at basic block i. The change in history pattern on executing $i \rightarrow j$ is given by $\Gamma(\pi, i \rightarrow j) = \pi$ if $i \rightarrow j$ is an unconditional jump. If $i \rightarrow j$ is a taken (non-taken) branch then $\Gamma(\pi, i \rightarrow j)$ is $left(\pi, 0)$ ($left(\pi, 1)$).*

In the popular *gshare* [4] scheme, the BHR is XORed with last n bits of the branch address to look up the prediction table. Usually, *gshare* results in

Pgm.	gshare		GAg		local	
	Mispred		Mispred		Mispred	
	Obs.	Est.	Obs.	Est.	Obs.	Est.
check	3	3	3	3	198	198
matsum	204	204	204	204	200	200
matmul	223	223	223	223	200	200
fdct	7	7	7	7	4	4
fft	3678	6165	3398	5175	4129	5154
isort	9687	9952	587	598	399	399
bsearch	9	9	9	10	6	7
eqntott	203	205	202	206	203	204

Table 1: Observed and estimated misprediction count with gshare, GAg, and local schemes.

a more uniform distribution of table indices compared to *GAg*. We define the index π as $\pi = history_m \oplus address_n(I)$ where m, n are constants, $n \geq m$, $\oplus$ is XOR, $address_n(I)$ denotes the lower order n bits of I's address, and $history_m$ denotes the most recent m branch outcomes (which are XOR-ed with higher-order m bits of $address_n(I)$). And,

$$\Gamma_{gshare}(\pi, i \rightarrow j) = \Gamma(history_m, i \rightarrow j) \oplus address_n(I)$$

In local schemes, the index π for branch instruction I is the least significant n bits of I's address, denoted $address_n(I)$ (n is a constant). Here π is independent of the past execution history of other branches. The update of π due to control flow is given by $\Gamma_{local}(\pi, i \rightarrow j) = address_n(J)$, where $address_n(J)$ denotes the least significant n bits of the last instruction J in basic block j.

Bounding Mispredictions Given the definition of π and Γ, we derive inflow and outflow constraints on the flow of π through the control flow graph to derive upper bounds on v_i^π. To bound m_i^π, we note the following. Suppose there is a misprediction of the branch in block i with history π. This means that certain blocks (maybe i itself) were executed with history π, the outcome of these branches appear in the πth row of the prediction table, and the outcome of these branches *must have created* a prediction different from the current outcome of block i. To model mispredictions, we therefore capture repeated occurrence of a history π during program execution with differing outcomes; we provide constraints to bound such occurrences. Details of our modeling appear in [5] and are ommitted here for space considerations.

Experimental Results We selected eight different benchmarks for our experiments. We assumed zero cache misses and a perfect processor pipeline with no stalls except for penalty due to misprediction of conditional branches. These assumptions, although simplistic, allow us to separate out and measure the accuracy of our estimation technique. We assumed that the branch misprediction penalty is 3 clock cycles (as in the Intel Pentium processor). We used the SimpleScalar architectural simulation platform in the experiments. By changing SimpleScalar parameters, we could change the branch prediction scheme for the experiments.

To evaluate the accuracy of our branch prediction modeling, we present the experiments for three different branch prediction schemes: *gshare*, *GAg* and *local*. Since finding the worst case input of a benchmark (which produces the actual WCET) is a human guided and tedious process, we only measured the actual WCET assuming a 4-entry prediction table. The results appear in Table 1. In this table, we have shown only the observed and estimated misprediction counts to enable clear understanding of the accuracy of our technique (which models the effect of branch prediction). Even though not shown here due to space shortage, the estimation accuracy was independent of the prediction table size. Our esti-

mation technique obtains a very tight bound on the WCET and misprediction count in all benchmarks except `fft`. The reason is that the number of iterations of the innermost loop of `fft` depends on the loop iterator variable value of the outer loops. This problem can be solved by providing expressions on the loop iteration counts instead of constants, as shown in [2].

Using CPLEX, a commercial ILP solver distributed by ILOG, on a Pentium IV 1.3 GHz processor with 1 GByte of main memory, our timing estimation technique requires less than 0.11 second for all the benchmarks with prediction table size varying 4–1024 entries.

One major concern with any ILP formulation of WCET is the scalability of the resulting solution. To check the scalability of our solution, we formulated the WCET problem for the popular *gshare* scheme with branch prediction table size varying from 4–1024 entries. Recall that in *gshare*, the branch instruction address is XOR-ed with the global branch history bits. In practice, *gshare* scheme uses smaller number of history bits than address bits, and XORs the history bits with the higher order address bits [4, 6]. The choice of the number of history bits in a processor depends on the expected workload. In our experiments, we used a maximum of 4 history bits as it produces the best overall branch prediction performance across all our benchmarks. As Figure 2 shows, the number of variables generated for the ILP problem initially increases and then decreases. With increasing number of history bits, number of possible patterns per branch increases. But with fixed history size and increasing prediction table size, the number of cases where two or more branches have the same pattern starts to decrease. This significantly reduces the number of ILP variables.

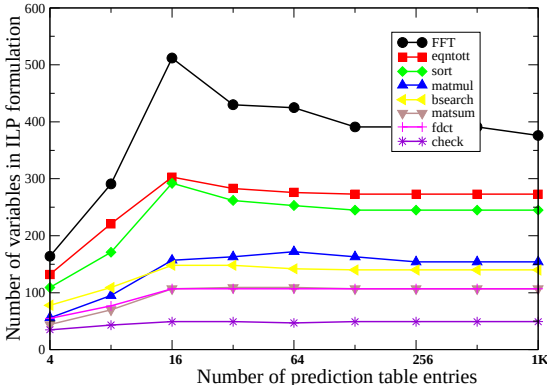


Figure 2: Change in ILP problem size with increase in number of entries in the branch prediction table for *gshare* scheme

References

[1] K. Chen, S. Malik, and D.I. August. Retargatable static software timing analysis. In *IEEE/ACM Intl. Symp. on System Synthesis (ISSS)*, 2001.

[2] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real time Systems*, May 2000.

[3] S-S. Lim, J.H. Han, J. Kim, and S.L. Min. A worst case timing analysis technique for in-order superscalar processors. Technical report, Seoul National University, 1998. *Earlier version published in IEEE Real Time Systems Symposium(RTSS) 1998.*

[4] S. McFarling. Combining branch predictors. Technical report, DEC Western Research Laboratory, 1993.

[5] T. Mitra and A. Roychoudhury. Effects of branch prediction on worst case execution time of programs. Technical Report 11-01, School of Computing, National University of Singapore, 2001.

[6] S. Sechrest, C-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *ACM International Symposium on Computer Architecture (ISCA)*, 1996.

Difficulties in Computing the WCET for Processors with Speculative Execution

Christine Rochange and Pascal Sainrat
Institut de Recherche en Informatique de Toulouse, France
{rochange, sainrat}@irit.fr

Abstract

In real-time applications, the Worst-Case Execution Time often needs to be estimated to check that deadlines will be respected. With the trend of using up-to-date processors, WCET computation techniques continuously have to evolve in order to take into account the most recent hardware features. In this paper, we show that ignoring speculative execution can lead to underestimated execution times, and we explain why modelling it is not straightforward. We feel that pure static analysis might not allow safe WCET computation, due to the fact that speculative execution prevents the decoupling between the high-level (path) analysis and the low-level (timing) analysis.

1. Introduction

For a large class of applications, embedded software has to satisfy hard real-time constraints. This requires to be able to tightly estimate the worst-case execution time (WCET) of programs.

WCET analysis has received much attention these ten last years. Dynamic methods involve measurements on real hardware or on cycle-level simulators. All the possible execution paths have to be explored in order to obtain the longest execution time. This poses two problems: (i) the number of possible paths is generally high and then the measurement time is prohibitive; (ii) for each path, the corresponding input data set has to be defined, which is usually difficult. In response to the drawbacks of dynamic methods, several static approaches have been proposed. They consist in three steps. First, the high-level analysis considers the program code in order to identify the possible execution paths, where a path is a list of basic blocks. Second, the low-level analysis estimates the execution time of each basic block. It is carried out in two phases: the global low-level analysis takes into account hardware components the behaviour of which depends on the global history of execution (e.g. cache memories); the local phase models components that only depends on the recent history (e.g. pipeline). Third, the execution times of paths are computed and the WCET is the longest one.

However, embedded systems tend to use modern processors featuring advanced architectural mechanisms that might be hard to model. Among these mechanisms, branch prediction, sometimes coupled with speculative execution, is implemented in most of the recent processors.

Estimating the WCET for processors with speculative execution does not present any special difficulty when it is based on dynamic measures: either the real target hardware is available (with speculative execution activated), or a cycle-level simulator is used, and speculative execution is not harder to model than other advanced features. However, current dynamic measurement methods often require to explore a too large number of execution paths and, for this reason, static analysis is generally preferred.

In this paper, we will show that estimating the WCET when a processor implements speculative execution is not straightforward. We suggest that usual static analysis techniques might not allow safe WCET computation, highlighting situations where they would lead to underestimation of the execution time.

Section 2 gives an overview of branch prediction and speculative execution techniques, and presents the work of Colin and Puaut [1] that takes branch prediction (but not speculative execution) into account within static WCET analysis. Section 3 shows why it is important to carefully model speculative execution to obtain a safe WCET. Section 4 discusses the difficulties of doing it within pure static WCET analysis, and section 5 concludes the paper.

2. Branch prediction and speculative execution

2.1 Overview

Modern processors are designed around longer and longer pipelines. Whenever a branch instruction is encountered in the instruction flow, the correct execution path is not known until the branch is executed. To avoid interrupting the instruction fetching, one of the two possible paths is speculatively selected by a branch predictor and instruction processing continues along this path. When the branch is resolved and if the speculative path is not the correct one, recovery actions are taken (e.g. the pipeline is flushed) and instruction processing restarts from the branch along the right path. Processing along a speculative path means fetching instructions from

the memory hierarchy, decoding and dispatching them to the reservation stations where they wait for their operands. For a processor that implements out-of-order execution, instructions belonging to the speculative path might also be executed before earlier instructions, and in particular before unresolved branches. This is what is called *speculative execution*. In that case, recovery from branch misprediction is a bit more complicated and generally requires mechanisms to restore the correct architectural state. Note that recovery is only required for components that must have a safe behaviour: the effects of a branch prediction error on other components, like cache memories or the branch predictor itself do not endanger correct functional results, they only might lower the system performance.

Many algorithms exist to predict the issue of branch instructions. The most recent ones include three kinds of structures:

- the *PHT* (*Pattern History Table*) is used to predict the direction of conditional branches: each of its entries reflects a recent history (often as a 2-bit saturating counter). The PHT is usually not tagged and it can be indexed by the instruction address (PC) alone or combined with a global or a local history recorded in one or several *BHR* (*Branch History Register)*. Thus, several branches share the same counter and, on the contrary, the behaviour of a branch depends on several counters according to the history.
- the *BTB* (*Branch Target Buffer*) is used to predict the target address (except for subroutine returns)
- the *RAS* (*Return Address Stack*) is used to predict subroutine returns.

2.2 Computing the WCET for processors with branch prediction

As far as we know, branch prediction has been considered within WCET analysis only for in-order processors: this work has been presented by Colin and Puaut [1]. They consider the Intel Pentium, which features a simple branch predictor based on a single table referred to as BTB. The proposed method includes several stages.

First, the control-flow graph is analysed to build an *abstract state* of the BTB for each basic block: it indicates which instructions might be contained in each entry of the BTB before and after the execution of the basic block. The *input* abstract state of a basic block is computed from the *output* abstract states of the possible preceding basic blocks. Then, the abstract states are used to classify the branch instructions and to determine, for each of them, if it will be correctly predicted or not. When ever this cannot be statically decided, the instruction is assumed to be mispredicted, which is supposed to be the worst case.

The WCET is then computed in two steps. First, a perfect branch predictor is assumed, and the WCET is estimated from the syntax tree and a set of formulas that express the maximum execution time of algorithmic structures. Then the timing effects of prediction errors are evaluated for a real branch predictor: a penalty delay is associated to each possibly mispredicted branch instruction. A second set of formulas is used to recursively build delay sets for each algorithmic structure of the syntax tree. The sum of these delays is then added to the WCET previously computed with perfect branch prediction.

3. Possible effects of speculative execution

When a processor implements speculative execution, processing along the wrong path may have two kinds of effects on the system. In this section, we describe these effects and show why ignoring them can lead to underestimate the WCET.

3.1 Dynamic instruction scheduling

Instructions of the wrong path occupy hardware resources, like functional units. Now, some flushing policies implemented for branch prediction error recovery do not immediately free the functional units. Thus, an instruction of the wrong path might continue its execution in a multi-cycle functional unit after the flushing of the pipeline (but its result will then be simply discarded). If the functional unit is not pipelined, the execution of later instructions belonging to the correct path might be delayed. Then the misprediction penalty would be longer than the strict recovery time.

Moreover, inserting wrong path instructions in the pipeline can modify the scheduling of previous instructions. For example, an instruction belonging to the wrong path that has its operands ready can be scheduled before a preceding instruction that is waiting for one of its operands. This might completely modify the overall scheduling, and then the execution time as mentioned in [3].

If speculative execution is not taken into account, the pipeline reservation tables produced by the local timing analysis might not be correct and the computed WCET could be underestimated.

3.2 Memory contents

Processing along the wrong path can also change the content of memories. If instructions of the wrong path miss in the instruction cache, they are fetched from the upper level of the memory hierarchy. This can have a detrimental effect on the program execution time if instructions of the wrong path replace in the cache instructions belonging to the correct path: when the execution later restarts along the right path, those replaced instructions will miss in the cache, thus requiring longer fetch times. This detrimental effect is often referred to as *cache pollution*. Note that fetching instructions along the wrong path can also have a beneficial effect, as reported in [4]: some instructions of the wrong path can later be found on the correct path, and then processing along the wrong path acts as a prefetch mechanism.

The same effects can be observed on data accesses, provided that instructions of the wrong path are executed (not only fetched), which is only allowed in dynamically-scheduled processors.

Processing along the wrong path may also have a beneficial or detrimental impact on the memories of the branch predictor (BTB, BHR, PHT and RAS) if they are updated speculatively in the earlier stages of the pipeline [2]. Only few parts are checkpointed for cost reasons (e.g. checkpointing the BTB is probably not affordable). If recovery is not implemented, the branch predictor tables might be polluted by the execution of the wrong path.

Now, let us assume that, ignoring speculative execution, the global low-level analysis is able to statically determine the real behaviour of all instruction and data cache accesses (hit or miss) and of all branches (well- or wrong- predicted). To understand why the possible pollution of memories due to wrong path execution should not be ignored, let us consider the following example:

```
for (i=0 ; i<10 ; i++)
{
      s[i] = 0;
      for (j=0 ; j<10 ; j++)
      {
            s[i] = s[i] + t[i][j];
      }
      m[i] = s[i] / 10;
}
```

This program may be compiled as:

```
L0    i=0;
L1    if i==10 then branch to L7
L3    s[i]=0
      j = 0
L4    if j==10 then branch to L6
L5    s[i] = s[i] + t[i][j]
      j++
      branch to L4
L6    m[i] = s[i] / 10
      i++
      branch to L1
L7
```

If we consider a branch prediction algorithm based on 2-bit saturating counters, initialised to "weakly-taken", the branch instructions of basic blocks `L1` and `L4` are mispredicted in the first iteration of loops `i` and `j`, and correctly predicted in the other iterations, while those of basic blocks `L5` and `L6` are always well predicted. As far as data accesses are concerned, the first reference to `s[i]` is determined to miss in the data cache while the other ones should hit.

Now, what does really happen if the processor implements speculative execution? At the first iteration of loop `i`, since the branch of basic block `L1` is mispredicted, some instructions belonging to the wrong path are processed. In particular, access to `m[i]` might be executed. If `m[i]` happens to fall in the same cache line as `s[i]` then, when the branch is resolved and the execution restarts along the correct path, `s[i]` misses in the data cache, contrarily to the conclusion of the global low-level analysis. As a result, the actual execution time might be longer than the estimated WCET, which is not acceptable.

In the same manner, the possible pollution of other memories (instruction cache, branch prediction tables,…) can increase the execution time. Ignoring speculative execution might again lead to an erroneous classification of instructions (branches or memory accesses) in the global low-level analysis step, which may result in an underestimated WCET.

4. Towards a safe WCET estimation for processors with speculative execution

We have shown why the execution of the wrong path has to be carefully taken into account in order to obtain a safe WCET. In this section, we discuss the difficulties of modelling speculative execution as part of static WCET analysis.

The possible effects of speculative execution on the dynamic scheduling of instructions can probably be taken

into account without excessive complexity. For example, the delaying of the execution of later instructions due to the occupation of hardware resources by wrong path instructions could be included in the WCET estimation by systematically adding to the misprediction recovery penalty the longest functional unit latency. An other solution would consist in assuming in-order instead of out-of-order execution, but it would lead to a very pessimistic WCET estimation.

The effects of speculative execution on the content of memories may be harder to take into account within a purely static WCET analysis. We have seen that it can invalidate the results of the global low-level analysis: memory accesses (either to instructions or data) classified as "cache hits" might actually miss due to the pollution of the cache by the execution of the wrong path; branches classified as "well predicted" might actually be mispredicted due to the pollution of branch predictor tables. This means that, in order to produce a safe classification of instructions, the global low-level analysis should take into account the instructions of the wrong path. Now, we feel that considering wrong paths within static analysis is not straightforward, since it probably requires new algorithms for syntax tree or control-flow graph traversal. Moreover, the number of instructions or basic blocks to include in a wrong path depends on the processor state (occupancy of hardware resources) and can only be determined during the local low-level analysis. Thus, it appears that a correct modelling of speculative execution would require a very close interaction between the high-level (flow) and low-level (timing) analyses, which are usually carried out independently.

While decoupling the analyses of different components (caches, branch predictor, pipeline, …) probably makes static WCET computation feasible, we wonder if the requirement of more interaction between these analyses to be able to take into account more and more advanced hardware features can be satisfied. If not, growing emphasis should be put on dynamic measurement (on real systems or simulators) to obtain accurate timing information while the static part of WCET estimation would focus on selecting the execution paths to explore with the goal of minimizing the measurement requirements.

5. Conclusion

A lot of work has been done these last ten years to allow static estimation of the WCET for processors with advanced features like cache memories, pipelined execution, branch prediction. The most recent dynamically-scheduled processors implement speculative execution: when a branch instruction is predicted, the instructions belonging to the predicted path can be executed before that the branch is resolved. In this paper, we discussed the possible effects of executing the wrong path whenever a branch is mispredicted.

We have shown that wrong path execution can modify the scheduling of the correct path instructions and/or change the content of memories (instruction and data caches, branch predictor tables, …). Then we have explained why ignoring these effects in WCET analysis can lead to an underestimated WCET, which can be dramatic for hard real-time systems.

We feel that usual static WCET computation techniques cannot accurately take speculative execution into account, since it would require a too complex interaction between global and local low-level analysis.

6. References

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

[2] S. Jourdan, T.-H. Hsing, J. Stark, Y. Patt, "The Effects of Mispredicted-Path Execution on Branch Prediction Structures", *Int. Conf. On Parallel Architectures and Compilation Techniques*, Octobre 1996.

[3] T. Lundqvist, P. Stenström, "Timing Anomalies in Dynamically Scheduled Processors", *20th IEEE Real-Time Systems Symposium,* December 1999.

[4] J. Pierce, T. Mudge, "Wrong-Path Instruction Prefetching", *IEEE Int. Symp. On Microarchitecture,* December 1996.

Why You Can't Analyze RTOSs without Considering Applications and Vice Versa

Jörn Schneider
Dept. of Computer Science, Saarland University, Germany
E-mail: js@cs.uni-sb.de

Abstract

Traditionally WCET analysis tools are designed for the analysis of application code. The execution time of RTOS (Real-Time Operating System) services and the interaction between RTOS and application are usually not considered. When performing an RTOS aware schedulability analysis the WCETs of RTOS services are needed. At first sight the application of existing WCET analyzers on RTOS code should be straightforward and should deliver the same accuracy as for application code. The paper explains why this is not the case, and why the presence of an RTOS diminishes the accuracy of application code WCET-analysis.

In addition to explaining why RTOSs should not be analyzed without considering application code and vice versa, the underlying problems are identified as well as enlightened by some examples and possible solutions are sketched. Eventually a comprehensive approach for WCET- and schedulability-analysis is proposed.

1 Introduction

Current WCET analyzers [6, 2] aim at analyzing application code. The execution time of RTOS (Real-Time Operating System) services and the interaction between RTOS and application are usually not considered. The schedulability analysis is expected to consider the overhead due to the RTOS. However, the schedulability analysis needs at least the WCET of relevant RTOS services. Colin and Puaut made a first attempt to apply static program analysis to an RTOS [1]. They compute the WCET of some RTEMS [3] system calls and report several problems in applying their WCET analysis. For instance the loop bound of the RTEMS scheduler could not be derived because it depends on the number of task arrivals during its execution (the scheduler loops until no further arrivals are noticed). The average WCET overestimation reported is 86%. The reported problems originate mainly from a methodical weakness of their approach. As they regard the RTOS isolated from the application.

This paper explains why RTOSs should not be analyzed without considering application code and vice versa. The underlying problems are identified as well as explained by some examples and possible solutions are sketched.

2 Analyzing RTOSs

2.1 What are the problems?

The WCET of RTOS services is highly dependent on the application using them. Table 1 gives a systematic list of such dependences. Examples of such dependences can for instance be found in the RTEMS code, and in the code of osCAN (an OSEK [4] implementation by Vector Informatik).

Determining factors of the WCET of RTOS services	Examples
Non-constant call parameters in application code	Any service with call parameter dependent control flow
Static, application dependent configuration parameters	Any service with loop bounds depending on no. of RTOS objects (e. g. tasks, resources)
Cache state	Replacement of RTOS owned cache sets by application code
Calling history of RTOS services	Scheduler execution after disabling preemption
Calling context	Call to system service from task, interrupt or operating system level

Table 1. Sources of WCET variations of RTOS services.

2.2 How can these problems be addressed?

Non-constant calling parameters When system calls are analyzed as part of the application, any knowledge about parameter values (e. g. obtained by a value analysis [2]) can be used to derive sharper bounds on the WCET.

Static configuration parameters The configuration parameters (e. g. number of tasks and memory mapping of tasks) are fixed for a particular application. Therefore, they can be considered either manually or automatically by WCET analysis as well as schedulability analysis.

Cache state If the cache is not partitioned in a special way, application code or data might displace cache sets occupied by the RTOS. No isolated WCET analysis of the RTOS can therefore benefit from positive cache effects caused by previous runs of RTOS services. It might even be impossible to consider the positive intrinsic cache effects of RTOS services. RTOSs are usually designed to minimize the number and duration of non-interruptible code sections. It is impossible for an isolated analysis to predict the negative impact of application interrupts outside these few code sections, unless all positive cache effects are ignored. In the case of a combined analysis it is possible to bound the effect of application caused cache replacements as it has been shown in [5] for the application analysis.

This is not only a question of the schedulability analysis. Depending on the application it can be beneficial, and for certain modern CPU types even necessary, to consider the preemption related cache effects within the WCET analysis (cf. [5]).

Calling history of RTOS services It not only affects the WCET of RTOS services, but often has an immediate impact on the task response time as well (e. g. an RTOS service called to disable preemption eliminates the subsequent interference by other tasks). A good schedulability analysis should consider these effects. Therefore, the history information should be statically predicted anyway and can also be used by WCET analysis.

Calling context When using a combined analysis, the calling context can easily be regarded. The WCET analysis can for instance consider infeasible paths for the specific calling context.

3 Analyzing applications

This section discusses problems arising in presence of multitasking RTOSs. The two subsection treat the problems in the same order. First the problem domain of data values is considered. Issues that arise due to isolated analysis of application and RTOS code come second. Not all of these issues can be addressed by a mere integration of application and RTOS WCET-analysis. The last parts of either subsection and Section 4 cope with this enigma.

3.1 What are the problems?

The data values used in application code can play a large role in computing the WCET.[1] Examples are: loop bounds, addresses of memory references and infeasible paths. For the WCET-analysis to profit from this fact a static prediction of value ranges is necessary. The value analysis described in [2] provides this functionality for instance. However, tools of this kind are—like any available WCET tools—designed for sequential programs. The following issues arise in presence of a multitasking RTOS:

Shared application memory Accesses by other tasks may change the value of data in such areas.

RTOS data structures Any RTOS data structure not unique to the analyzed task might be changed by RTOS services called in other tasks. Even data structures unique to a task might be manipulated by other (user or RTOS) programs.

Memory mapped I/O The values read from those areas are mainly determined by the environment and accesses are non-cachable.

The WCET analysis of application code should consider the WCET of the system calls used. A seemingly attractive approach is to initially ignore the system calls within the application WCET analysis and thereafter add system call WCETs obtained by an isolated analysis of the RTOS. However, there are good reasons not to do so (see Table 2).

There is a significant difference between problem descriptions 1 through 3 and the classes of problems alluded to by description 4 of Table 2. The former difficulties occur also together with the isolated WCET-analysis of library functions, the latter not.

3.2 How can these problems be addressed?

Because of shared application memory and RTOS data structures, a value analysis has either to ignore such data, or has to be enhanced to a multi-task-analysis. The latter is not trivial. Memory mapped I/O areas have to be excluded from the value analysis since they are volatile.

[1] This holds for RTOS code also. Nevertheless the subject is discussed in this section because that is where WCET-analysis comes from and because applications are usually more data-driven than RTOSs.

No.	Problem description
1	RTOS WCETs are systematically overestimated (shown in Section 2)
2	Information about correlation of worst-case paths and number+context of RTOS calls is destroyed $\Rightarrow$ only a pessimistic approach can still deliver conservative WCETs
3	Cache and pipeline effects caused by RTOS calls cannot be considered in application WCET
4	It is impossible to consider positive effects of concepts existing only in presence of multi-tasking RTOSs, for instance RTOS calls dynamically raising the application priority (e. g. by disabling preemption or interrupts, or by occupying resources)

Table 2. Problems of analyzing applications isolated from the RTOS.

Section 2 shows that the WCET of RTOS services depends on the *call situation*. This call situation subsumes the factors given in Table 1. Some aspects of the call situation are not unique to applications running on RTOSs. These aspects can be identified already, when stand-alone applications with calls to library routines are considered (one could replace the word *RTOS* with *library* in the problem descriptions number 1 through 3 of Table 2 and the statements would still be true to some extent). Aspects like these can be addressed by embedding the analysis of library/system calls within the application WCET analysis. The embedding can be implicitly or explicitly. Embedding implicitly means that the calls are treated like an ordinary function call. The input data structure (e. g. the control flow graph) of the WCET analyzer contains all needed information to analyze such calls. If the WCET analyzer uses machine code as input (e. g. the one described in [2]), this can even be done without providing the user with the library/RTOS source code. Embedding explicitly means that two independent WCET analyzers (or two instances of the same analyzer) are used, one for the application and one for the RTOS. The RTOS WCET analyzer can be a black box that takes the code of the RTOS service as well as collected information about calling parameters, static configuration parameters, cache state and calling context as input (see Subsection 2.2).

However, introducing an RTOS in the considered scenario adds completely new qualities to the problem of WCET analysis (represented by item no. 4 of Table 2). These are issues that cannot be addressed by a mere integration of application and RTOS WCET analysis. Rather a high-level view is needed to consider them in WCET and schedulability analysis.

Several such high-level concepts can be identified that co-determine the temporal behavior of the tasks of an RTOS-based system. These concepts are for instance the effective priority of tasks, the RTOS mode (e. g. initialization or normal operation mode), application modes, task states, and the system level (task, interrupt or RTOS level). Those high-level concepts have a certain meaning when viewing the system as a whole rather than as a bunch of independent programs at a microarchitectural level. At run-time the properties of these concepts have a defined state at each point in time. We define the *meta-state* of a task to be the set of these states.

In a static approach, at best partial knowledge of the meta-state of a task can be obtained. To address the RTOS specific problem domain, partial knowledge can be collected which allows to compute the worst-case response time of tasks more accurately. This includes exploiting meta-state information to compute sharper bounds on WCETs of tasks as well as sharper bounds on microarchitecture-related preemption costs.

4 Proposal for a comprehensive WCET- and schedulability-analysis approach

The authors present work on a comprehensive WCET- and schedulability-analysis approach exploits meta-state information to compute sharper bounds on WCETs of jobs (tasks and interrupt service routines) and interesting code sections and on microarchitecture-related preemption costs. The framework exploits the following aspects of the meta-state of a job: effective priority of a job (determined by: locked interrupts, preemption lock, occupied resources), RTOS mode and current system level. The meta-state information is exploited as follows:

1. Extrinsic cache effects are considered by the cache analysis (which is a part of the WCET analysis) in dependence of the effective priority at each program point of the analyzed job.

2. Pipeline-related preemption costs are individually computed for each job, again in dependence of the effective priority, and are considered during the schedulability analysis.

3. The WCET of jobs and code sections is computed for the proper RTOS mode (initialization mode or normal operation mode) and for each RTOS mode a separate schedulability analysis is undertaken.

4. The current system level is considered when the WCET of system calls is computed.

Similar to the cache- and pipeline-sensitive schedulability analysis described in [5] the cache-related preemption

costs are incorporated in the WCET while the pipeline-related preemption costs are explicitly considered during the schedulability analysis.

The above sketched framework uses the WCET analysis tool described in [2]. The WCET analyzer is loosely coupled with the surrounding tools. It is guided with the help of the obtained meta-state information in order to compute sharper bounds on the WCET and the microarchitecture-related preemption costs. A detailed explanation of this method is beyond the scope of this paper.

5 Conclusion

The paper showed that analyzing WCETs of RTOS-based real-time systems—whether of RTOS services or of application code—requires other than the established approaches. The underlying problems were explained by examples and classified. Additionally it was sketched how the individual problems can be addressed. Finally a comprehensive approach for WCET- and schedulability-analysis was proposed. The proposed approach shows how it is possible to overcome most of the obstacles obstructing the path toward comparative results of WCET-analysis for RTOS-based systems.

Acknowledgements

Many members of the compiler design group at the Universität des Saarlandes, especially the members of the USES (Universität des Saarlandes Embedded Systems) group, deserve acknowledgement. Reinhard Wilhelm, Daniel Kästner, and Stephan Diehl carefully read draft versions of this work and provided many valuable hints and suggestions.

I would like to thank the anonymous reviewers for their helpful comments.

References

[1] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, June 2001.

[2] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, Lake Tahoe, USA, Oct. 2001.

[3] On-Line Applications Research Corporation, Huntsville, AL, USA. RTEMS Applications C User's Guide. Edition 4.0, Oct 1998. `http://www.oarcorp.com/RTEMS/rtems.html`.

[4] OSEK/VDX – Open systems and the corresponding interfaces for automotive electronics. OSEK/VDX Operating System. Version 2.2, Sept. 2001. `http://www.osek-vdx.org`.

[5] J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium 2000*, pages 195–204, Nov. 2000.

[6] J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34 of *ACM SIGPLAN Notices*, pages 35–44, May 1999.

Is Worst-Case Execution-Time Analysis a Non-Problem? — Towards New Software and Hardware Architectures *

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

Abstract

Despite the scientific advances in the worst-case execution-time (WCET) analysis, there is hardly any industrial impact of the research solutions presented so far. This seems to be due to the high complexity of implementing and using the proposed WCET approaches.

This paper discusses what makes WCET analysis complex and proposes to use adequate hardware and software architectures to improve the predictability of program timing, thus simplifying WCET analysis.

1 Introduction

Research in worst-case execution-time (WCET) analysis has been around for one and a half decades. During this period a number of different approaches to WCET analysis, including solutions for modelling hardware features and characterizing possible execution paths of real-time tasks have been found [5]. Still, the results of WCET-analysis research have hardly any impact on the industrial practice of timing analysis. This seems to be due to the high complexity of the implementation and use of WCET analysis tools. In addition, WCET research always seems to lag one step behind the advances in microprocessor technology – whenever WCET research manages to deal with the features of one hardware generation the next generation of processor and hardware architectures, equipped with novel speedup features, are already there.

This paper proposes to use new, adequate hardware and software architectures to improve the temporal predictability of programs, and thus reduce the complexity of WCET analysis. Hardware architectures for future real-time systems must allow to determine instruction execution times locally by inspecting single instructions and only a small number of instructions preceding them. Software architects have to investigate into programming techniques that reduce the number of input-data dependent branching decisions in the software, thus reducing the number of different execution paths of a program. The further sections presents our considerations in more detail and proposes possible solutions.

*This work has been supported by the IST research project "High-Confidence Architecture for Distributed Control Applications (NEXT TTA)" under contract IST-2001-32111.

2 The Complexity Dilemma of WCET Analysis

There is no doubt that WCET analysis as it is currently used is a complex problem. It has been shown that, in general, the number of paths to be analyzed for an exact WCET analysis of a piece of code grows exponentially with the number of consecutive branches in the control flow of the analyzed code. This statement assumes that the code (a) is coded in traditional style (i.e., not applying programming techniques that focus on ease of WCET prediction) and (b) is to be executed on a modern high-performance processor architecture that includes caches and pipelines. Except for very simple programs this high complexity makes the full path enumeration needed for an exact WCET analysis intractable [3].

Current approaches to WCET analysis deal with this complexity in two ways:

- Calculate a high-quality WCET bound by accepting long computation times for the analysis.
- Trade the feasibility or speed of analysis for quality, by using simplified but pessimistic models of the possible software behaviours and the hardware timing.

The dilemma of WCET analysis is that neither of these approaches is acceptable in a commercial setting. On the other hand, using current hardware and software architectures does not allow for a better solution – the complexity of the problem simply is there. This raises the question if the current approaches to WCET-analysis are the correct answer to the problem of task timing analysis, or if current WCET research is focussing on the wrong problem.

2.1 Sources of Complexity

Puschner and Burns [5] identified two central factors that determine the WCET of a program in a given application context, (a) the possible sequences of program actions in a given application, and (b) the time needed for each action in each of these possible sequences. Clearly, both factors do not only determine the WCET of the code, but also the complexity of WCET analysis. Possible sequences of actions (instructions) depend of the algorithm that has been chosen to implement a solution to a problem and the code manipulations the compiler performs during compilation. The time needed for each action (instruction) depends on the features and configuration of the machine (hardware) on which the actions are executed. A number of principles applied in typical modern hardware and code design can be made responsible for WCET complexity. In the following we focus on two such principles, one for hardware and one for software.

Hardware Speedup by Speculation: This is the principle found in hierarchical memory architectures, e.g., cache. Instructions or data are loaded into (and kept in) small buffers (caches) with short access times — these access times are typically much shorter than the access times of the larger store that holds larger portions of instructions and data — with the intention to speed up future memory accesses. The decisions about which items are to be loaded, kept, and replaced in cache are usually guided by heuristics, i.e., speculation about which items might be accessed in the near future.

The use of speculative decision mechanisms leads to variable memory access times. The duration of each particular memory access, in turn, depends on the state in which the preceding operations have left the cache. Both effects (the fact that memory access times vary and the dependency of actual memory access times on the execution history) taken together contribute to the complexity of WCET analysis.

Software Optimization for Frequent Scenarios: Real-time programmers use algorithms and programming techniques that have proven to be effective for non real-time applications. In non real-time applications, speed optimization for the most probable (i.e., frequent) scenarios is the primary goal. Temporal predictability is not an issue. In order to favour frequent cases, non real-time algorithms choose the actions to be performed based on input data. Input-data dependent control decisions, however, cause programs to execute on different execution paths with different execution times. As a consequence the number of different cases to be considered by the WCET analysis is potentially high.

3 Possible Ways Out of the WCET Dilemma

This section illustrates the potential of alternative hardware and software architectures to simplify WCET analysis significantly. It provides an alternative to each of the two mentioned design principles.

3.1 Hardware Speedup: Control Instead of Speculation

In contrast to non real-time applications, (hard) real-time applications primarily require temporal predictability. Appropriate hardware designs therefore support WCET analysis via predictability. This can be achieved by using memory hierarchies that exercise absolute control on the contents of fast buffers instead of relying on speculation. Rather than hoping that future memory accesses result in a cache hit, adequate prefetching strategies make the contents and thus access times of high-speed memory easy to predict. A memory architecture that achieves predictability by prefetching has been proposed a number of years ago [2]. Unfortunately, alternative memory architectures have not been further explored.

3.2 Software: Getting Rid of Input-data Dependencies

The second problem we mentioned is that traditional algorithm design and optimization yields code that behaves differently for different input data. To circumvent this problem and allow for a simple analysis, program behaviour must be less dependent on input-data values. By reducing input-data dependencies the number of paths to be considered during WCET analysis gets smaller and, as a consequence, the complexity decreases.

Following this concept we developed the single-path paradigm [6]. The single-path paradigm yields programs that are fully temporally predictable. The central idea of the paradigm is to generate programs whose behaviour is completely independent of input data and which thus always execute on the one and only possible execution path.

Single-path programming builds upon a code transformation that removes data-dependent branching statements from the code. This code transformation is capable of transforming every WCET-analyzable piece of code into code with a single path. The transformation uses two different strategies to convert statements with if-then-else and loop semantics, respectively. If-then-else and other sequential branching statements with an input-data dependent branching condition are transformed into strictly sequential code by using *if-conversion*, [1]. Loops with input-data dependent termination are converted into loops with a constant — the maximum — iteration count. The termination condition of such loops is built into the head of a new *if* statement that is generated in the body of the loop being transformed. As a last step, if-conversion is applied to the newly generated *if* statement, see [4].

The fact that programs only have a single execution path makes WCET analysis trivial: First, path analysis is superfluous: observing the execution path of any code execution with any input data yields

the singleton execution path. Second, the analysis does not need complex and accurate hardware timing models for static WCET analysis. Since programs following this paradigm only have a single path, this singleton path is necessarily the worst-case path. Thus, obtaining the WCET by measurements is possible (either by measurements on the target or on a cycle-accurate hardware simulator) and there is no need to build any specific tools for static analysis. The latter also provides a solution to dealing with new hardware features in the analysis (see above). As the WCET analysis of single-path programs does not require hardware modelling, software developers do not have to wait until tool vendors incorporate the new features into their models in order to perform WCET analysis for their new platforms.

4 Conclusion

"Is WCET analysis a non-problem?" is the question posed in the title. To answer this question we investigated whether highly sophisticated WCET analysis techniques are the correct way to deal with the complexity of task timing analysis. We discussed hardware and code design practices that cause complexity and proposed an alternative memory architecture and the single-path programming paradigm as possible ways out.

The answer to the original question seems to be "Yes and No": As long as real-time code is coded for speed rather than temporal predictabiliy and hardware manufacturers continue to use memory hierarchies that rely on speculation then the answer is "no" — and we will certainly have to deal with such systems for at least one more decade. On the other hand, if people get aware of the importance of temporal predictability and build systems correspondingly, then WCET analysis indeed becomes trivial. So the new question is if it will be possible to convince people to change their way of thinking and put temporal predictability first.

Acknowledgments

The author would like to thank Raimund Kirner for his valuable comments on an earlier version of the paper.

References

[1] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[2] M. Lee, S. Min, C. Park, Y. Bae, H. Shin, and C. Kim. A Dual-mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times. In *Proc. 14th Real-Time Systems Symposium*, pages 98–105, 1993.

[3] T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium*, pages 12–21, Dec. 1999.

[4] P. Puschner. Transforming Execution-Time Boundable Code into Temporally Precdictable Code. In *Proc. IFIP World Computer Congress, Stream on Distributed and Parallel Embedded Systems*, Aug. 2002.

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

[6] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

How much Worst Case is Needed in WCET Estimation? *

Stefan M. Petters
Department of Computer Science
University of York
United Kingdom
Stefan.Petters@cs.york.ac.uk

Abstract

Probabilistic methods provide probability density functions for the execution time or the assumed worst case execution time instead of a single WCET value. While the resulting probability tends to fall towards zero quickly, the actual zero value, i.e. the 100 % guarantee, is reached only with unreasonable overestimation of the real WCET. In order to cope with this, this paper proposes to use similar techniques to hardware dependability analysis, where a 100 % guarantee is physically impossible and a certain, usually very small amount of risk is acceptable.

1 Motivation

Modern high performance processors include many features which usually make cycle true simulation infeasible. As a result this either imposes impractical limitations on the code or operating system to allow for WCET estimation, or the methods used to capture these effects have to introduce simplifications that lead to results which may be up to an order of magnitude beyond the physical possible WCET.

One possibility to get around this problem is the deployment of statistical methods. Throughout this paper the validity of these methods is assumed. Additionally it has to be assumed that the methods provide a description of the program behaviour which correctly covers the execution time for all modi of operation. A major problem of such approaches is that they result in approximations of the (worst case) execution time, whose probabilities are non-zero, but very small for a long way beyond the physical WCET. The question now is, whether one has to go for the zero-probability of an error, which tends to be as pessimistic as static WCET analysis, or if a probabilistic guarantee suffices. While the first case has no real advantage compared to static WCET analysis, the second has the open issue as to which probabilistic guarantee to accept as good.

2 Probabilistic WCET Analysis

Research in probabilistic WCET analysis can be divided in two categories:

- Appraches using observed test cases to reason about the probability of an execution time not observed during the tests.
- Approaches analysing small parts of the program in order to reason about the probability of different combinations of the results of the smaller units.

*The work presented in this paper is supported by the *European Union* as part of the research programme “Next TTA”

As there are currently no publications in the second area, we will focus on an example of the first research area. Stewart Edgar uses a black box approach in [1]. The description of the method here can only be very coarse and the reader should have a look at the original papers on this topic (e.g. [1, 2]).

The program is run several times, with random input data and the end-to-end execution time of the program runs are measured. As it is obvious, the measurements will not likely include the physical WCET of the program on that processor in the general case, extreme value statistics are deployed to reason about the execution time longer than any experienced during measurement. Extreme value statistics are concerned with modelling the right and/or left hand tail of a probability distribution, as opposed to the modelling of the average case with conventional statistics. This induces that outliers in the measurement data, which are usually disregarded with conventional statistics, have considerable impact on the modelling parameters of extreme value probability density functions.

Extreme value statistics are well known in the area of financial risk assessment and civil engineering. In the latter case the assessment of maximum wind speeds or flood levels is computed utilising these technique in order to dimension the statics of buildings. There are three type of extreme value probability density functions, described with a theorem which corresponds to the central limit theorem of the normal distributions. As a necessary precondition to apply this technique, the underlying random variables have to be independent and identically distributed.

For the approach the most simple solution of a *Gumble distribution* has been chosen. This model only uses deviation and mean of the random variable, in our case the observed execution time. The following equation show the Gumble probability density function and the cumulative Gumble probability density function:

$$g(t) = exp\left(-\left(e^{-\frac{t-\mu}{\sigma}} + \frac{t-\mu}{\sigma}\right)\right)\frac{\mu}{\sigma} \quad (1)$$

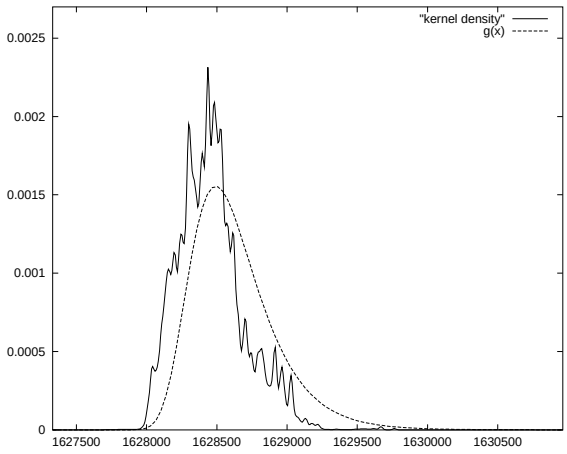


Figure 1: Sample Measurement Data and Extreme Value Approximation.

$$G(t) = exp\left(-e^{-\frac{t-\mu}{\sigma}}\right) \quad (2)$$

The cumulative variant expresses the probability of an execution time below the value t.

An example execution time measured and the corresponding Gumble distribution is given in figure 1. The measured times are given in a kernel density transformed representation. The transformation is used to display discrete data as a continuous curve and thus allowing the comparison by inspection with the extreme value approximation.

A major drawback of using the Gumble distribution to approximation is the non-zero probability for execution times, except for $\pm\infty$. While the probability of the execution time exceeding 20σ beyond the mean is only $2.06E^{-9}$, the "risk" is still there. As experiments show, this probability reaches quickly $1.0E^{-20}$ and less with an overestimation of some 10% (cf. [3]).

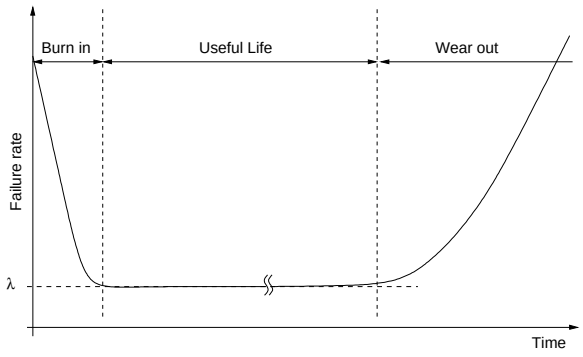


Figure 2: Typical Variation of Fault Ratio of Hardware Components over Time [4].

3 Hardware Considerations

This section will give a short introduction in the mechanisms to risk assessment of hardware components. Figure 2 shows the typical distribution of hardware faults in electronic equipment over time.

During the period of *burn in* the probability of hardware failure is higher, due to faults in the productions of the components. A good example for such a behaviour are errors due to the statistic deviation in the doting of semiconductors. To avoid the high probability of failures in the *burn in* period, the components are in general case run for a time before deployment in a dependable system to weed out bad components. This process is in most cases speeded up by undertaking this testing phase under more extreme circumstances than the system has to endure in real operation (e.g. heat, cold, mechanical stress). Thus a production error that might show up only after months or years down the line is uncovered after a few hours or days of operation.

After the burn in time, the hardware components reach a more or less constant failure rate of λ. Usually this useful lifetime is quite long. In the end the wear out sets in, where, for example, saturation effects[1] in the semiconductor set in. The failure rate λ after burn in as well as the average life time of a given hardware component is usually known. The reliability $R(t)$ of a component not to fail is given in equation 3.

$$R(t) = e^{-\lambda t} \tag{3}$$

For the computation of system failure usually the *mean time to failure* (i.e. λ) is taken to compute the overall systems failure rate. Since the usual failure rate is less then one failure in the lifetime (T_{life}) of a product, the failure rate can be transformed into a failure probability (p_{life}) for the lifetime of the system. This failure probability can be computed using equation 4.

$$p_{\mathsf{life}} = \int_0^{T_{\mathsf{life}}} R(t)dt \tag{4}$$

A similar reasoning may also be applied to software components. The major difference between software and hardware components is the discrete nature of failures of the software components as opposed to the continuous nature of failures of the hardware components. Assuming the program has no algorithmic errors, exceeding a computation time alloted to the program can be considered a software failure in real-time systems. A basic requirement for this is the assumption that the probability for an overrun of the alloted time for an individual run p_{excess} is known and constant for all runs. Additionally the maximum amount of task releases for any given time is essential for the computation. This is usually defined as a minimal inter arrival time T_{task}

The probability of a failure over the lifetime of the product is computed using equation 5.

$$p_{\mathsf{life}} = 1 - (1 - p_{\mathsf{excess}})^{\frac{T_{\mathsf{life}}}{T_{\mathsf{task}}}} \tag{5}$$

Defining an acceptable failure probability during the lifetime, which would be in the order of magnitude of

[1] One problem in the semiconductor industry is that the doting of semiconductors may be done by diffusion and these donated atoms tend to start drifting inside the semiconductor.

the failure probability of an hardware failure, it is easy to compute an acceptable p_{excess} transforming equation 5 into:

$$p_{\text{excess}} = 1 - p_{\text{life}}^{\frac{T_{\text{task}}}{T_{\text{life}}}} \quad (6)$$

4 Conclusion

While the number of publications in the area of probabilistic WCET estimation is quite limited up to now, the number of people working on this issue is becoming larger. Interpreting an overrun of an assumed value for the WCET of a program as a software fault, similar probabilistic techniques as for hardware component failures may be used. This is particular useful whenever probabilistic methods are utilised to reason about the WCET, as these methods tend to provide probability density functions to describe the WCET instead of a single value. The validity and applicability of this method is subject to discussion.

References

[1] A. Burns and S. Edgar, "Statistical analysis of WCET for scheduling," in *Proc. of the IEEE Real–Time Systems Symposium (RTSS'01)*, (London, United Kingdom), Dec. 4–6 2001.

[2] A. Burns and S. 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] S. 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.

[4] N. Storey, *Safety-Critical Computer Systems*. Addison–Wesley Publishing Company, 1996.