Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)

Chair:
Frank Mueller
North Carolina State University, USA

Dresden, Germany, July 4, 2006

Preface

On the 4th of July, 2006, the 6th International Workshop on Worst-Case Execution Time Analysis (WCET'06) was held in Dresden, Germany, co-located with the 18th Euromicro International Conference on Real-Time Systems (ECRTS'06), both with support of Euromicro Technical Committee. The goal of the workshop is to bring together people from academia, tool vendors and users in industry that are interested in all aspects of timing analysis for real-time systems. The workshop will provide a relaxed forum to present and discuss new ideas, new research directions, and to review current trends in this area. The workshop will be based on short presentations that should encourage discussion by the attendees.

The topics of the workshop include any issue related to timing analysis, in particular:

- Different approaches at computing WCET
- Flow analysis for WCET
- Low-level timing analysis, modeling and analysis of features
- Calculation methods for WCET
- Strategies to reduce the complexity of WCET analysis
- Integration of WCET and schedulability analysis
- Evaluation and case studies
- Testing Methods for WCET analysis
- Tools for timing analysis
- Design for Timing Predictability
- Integration of WCET analysis into the development process
- Compiler optimizations for worst-case paths
- WCET analysis for multi-processors, multi-cores or SMTs
- WCET analysis for networks (e.g., CAN)

WCET'06 featured one invited talk, one report of an upcoming WCET tool contest and, most of all, presentations of technical paper combined with discussions with the attendees. The papers were selected based on peer reviews by program committee members and outside reviewers, all experts in the field.

Acknowledgments

The workshop chair would like to acknowledge the following people:

- the invited speaker, Tullio Vardanega, Univ. of Padua (Italy), for his voluntary contribution to the workshop;
- Herman Härtig for his local support;
- the WCET SC for their advice;
- and last but not least the eager members of the program committee and the anonymous external reviewers.

The Chair,

Frank Mueller

June 2006

WCET'06 Program Committee

- Henk Corporaal, TU/e (Eindhoven University of Technology). Netherlands.
- Niklas Holsti, Tidorum Ltd.. Finland.
- Björn Lisper, University of Mälardalen. Sweden.
- Stefan Petters, National ICT Australia Ltd. Australia.
- Isabelle Puaut, IRISA Rennes. France.
- Jan Staschulat, University of Braunschweig. Germany.
- Gerhard Unterweger, Consultant for Automotive Industry. Germany.

WCET'06 Steering Committee

- Guillem Bernat, University of York. England, UK.
- Jan Gustafsson, Mälardalen University, Sweden.
- Peter Puschner, Technical University of Vienna, Austria.

Table of Contents

Algorithms for Infeasible Path Calculation

Jan Gustafsson, Andreas Ermedahl, and Björn Lisper
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson, andreas.ermedahl, bjorn.lisper}@mdh.se

Abstract

Static Worst-Case Execution Time (WCET) analysis is a technique to derive upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component in static WCET analysis is to derive flow information, such as loop bounds and infeasible paths.

Such flow information can be provided as either as annotations by the user, can be automatically calculated by a flow analysis, or by a combination of both. To make the analysis as simple, automatic and safe as possible, this flow information should be calculated automatically with no or very limited user interaction.

In this paper we present three novel algorithms to calculate infeasible paths. The algorithms are all designed to be simple and efficient, both in terms of generated flow facts and in analysis running time. The algorithms have been implemented and tested for a set of WCET benchmarks programs.

1 Introduction

To give timing guarantees for embedded and real-time systems, a key parameter is the *worst-case execution time* (WCET) of the software. A *static WCET analysis* analysis finds an upper bound to the WCET of a program, relying on mathematical models of the software and hardware involved. Given that the models are correct, the analysis will derive a timing estimate that is safe, i.e., greater than or equal to the WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as the execution time of individual instructions, as well as the program's *possible execution flows*, to bound the number of times the instructions can be executed, needs to be derived. The latter includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc.

This research has been supported by the KK-foundation through grant 2005/0271.

The goal of *flow analysis* is to calculate such *flow information* as automatically as possible. Flow analysis research has mostly focused on *loop bound* analysis, since upper bounds on the number of loop iterations must be known in order to derive WCET estimates [8].

Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and the possible inputs. Information on infeasible paths is not necessary to find a WCET estimate, but may tighten it.

This article presents ongoing work to automatically calculate infeasible paths. Three new and complementary algorithms are presented. They have been implemented in our prototype WCET analysis tool and tested for a set of WCET benchmarks programs.

The concrete contributions of this article are:

- We present ongoing work to extend our flow analysis method, called *abstract execution*, to calculate information about infeasible paths.
- We present three algorithms, calculating different types of infeasible path information, allowing us to trade analysis time for flow information precision.
- We show how to make our infeasible path algorithms input data dependent, allowing us to calculate more precise flow information for a program with limitations on its possible input data values.
- We evaluate the effect of our different infeasible path detection algorithms, including the type and amount of flow information generated.

The rest of this paper is organized as follows: In Section 2, we discuss causes of infeasible paths and describe related work. In Section 3, we describe our research prototype, SWEET. Section 4 describes the different algorithms, and Section 5 presents an illustrating example. Section 6 presents analysis results, and in Section 7 we draw some conclusions and discuss future work.

2 Causes of Infeasible Paths and Related Work

There are two different causes to infeasible paths. The first cause is semantic dependencies that always hold, as illustrated by the following code fragment:

`if` (`x` $<$ `0`) **A** `else` **B**; `if` (`x` $>$ `2`) `then` **C** `else` **D**

Here, both true-branches for the `if` statements are always in conflict[1], and the corresponding path **A-C** can never be taken.

A second cause to infeasible paths is due to limitations of input data values. Such limitations can be used to further limit the set of feasible paths. For example, if we know that `x>5` when the above code is executed, then we can conclude that the paths **A-C**, **A-D**, and **B-D** are all infeasible i.e., we find more infeasible paths with this additional knowledge.

Recent industrial WCET case-studies [7, 9, 17], have shown that it is important to develop good support for both loop bound analysis and infeasible path detection, thereby reducing the need for manual annotations. The case studies also showed that a mode- (giving a WCET estimate under certain system conditions) and input-sensitive WCET analysis often was preferable, in order to obtain better resource utilization and provide a better understanding of the system's timing characteristics. Thus, it should be important to develop input-sensitive infeasible path analyses.

There has been some work on automatic detection of infeasible paths for WCET analysis. Altenbernd [2] uses a combination of path enumeration, path pruning, and symbolic evaluation to find infeasible paths. Kountouris [13] studies detection of infeasible paths in the synchronous real-time language SIGNAL. Liu et al. [14] use symbolic evaluation of higher languages to avoid infeasible paths. Lundqvist and Stenström [15] find loop bounds and infeasible paths by symbolic simulation on the binary code. Healy et al. use value-dependent constraints to find infeasible paths [12]. Aljifri et al. [1] generate only the feasible paths using the concept of partially-known variables. Chen et al. [4] proposed a method that finds infeasible paths by identifying conflicts between assignments and branches, and between different branches.

The proposed infeasible path detection algorithms all use our flow analysis method *abstract execution* [10, 11], which is briefly described in the next section. This method has some similarities with the one of Lundqvist and Stenström [15], as well as with trace partitioning [3]. However, abstract execution uses a more detailed value domain, and it is based on an abstract interpretation framework.

[1]We assume, for simplicity, that the value of `x` is not modified in **A** and **B**.

3 SWEET and Abstract Execution

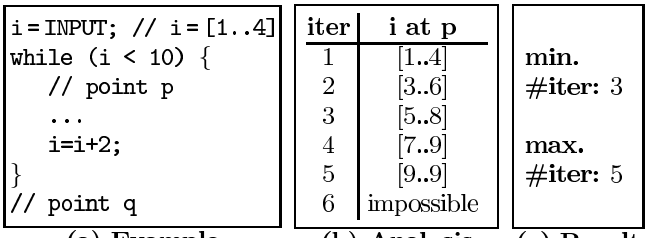
SWEET (SWEdish Execution time Tool) is a prototype WCET tool developed at Uppsala and Mälardalen University [16]. It consists of three main parts; a flow analysis which detects program flow constraints, a low-level analysis, where timing for program parts are obtained [6], and a final calculation where the longest execution path is extracted given information derived in the two preceding stages [8].

The current flow analysis of SWEET uses a *scope-graph* [8]. Each *scope* in the scope-graph is a different execution environment of a program, such as a function or a loop. See Figure 3 for an example. Our current scope-graph representation is context-sensitive, i.e., each call to a function or a loop in a function generates a different scope. Different calls to a function are analysed separately, which may yield higher precision but also a costlier analysis.

Abstract execution is a form of symbolic execution [10, 11], which is based on abstract interpretation. Rather than using traditional fixed-point iteration [5], abstract execution executes the program in the abstract domain, with abstract values for the program variables, and abstract versions of the operators in the language. For instance, the abstract domain can be the domain of intervals: each numeric variable will then hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. As usual in abstract interpretation, the abstract value held by a variable, at some point, represents a set containing the actual concrete values that the variable can hold at that point.

With abstract values, conditionals cannot always be decided, and the abstract execution must then execute both branches. In order to curb the growing number of paths, *merging* of abstract values for different paths can take place. A merged abstract value then surely contains all the possible concrete values from both paths, and a single-path abstract execution, representing the execution of both paths, can continue from the merging point. Typical merge points are places where different program flows meet, like after if-statements or loops. Merging may yield abstract values that represent the possible set of concrete values in a less precise way: for instance, the merge of the intervals [6..6] and [10..11] is [6..11], which also contains the concrete values 7, 8, 9 not present in the original intervals.

SWEET currently supports abstract execution with intervals. It allows the user to control the placement of merge points, in order to explore different tradeoffs



(a) Example (b) Analysis (c) Result

Figure 1. Example of abstract execution

between analysis speed and precision. Currently, the user can specify merge points to be one or more of the following types: after if-statements, after loop bodies, after loop exits and after function exits.

Figure 1 gives a simple example of abstract execution with intervals. The loop in Figure 1(a) is abstractly executed in Figure 1(b). As iteration 4 and 5 are executed, the set of possible values of `i` is reduced until, finally, the set of values for the true branch of the loop condition is empty, the loop condition is evaluated to FALSE only, and the abstract execution of the loop terminates. During the abstract execution, we keep track of the iteration count of the loop body, and Figure 1(c) shows the resulting loop bounds.

The abstract interpretation framework guarantees that a calculated abstract value always represents the set of possible concrete values. Thus, no execution paths will be missed by the analysis. On the other hand, an abstract value may overestimate this set, which means that the analysis may yield program flow constraints that are not tight. This means that some infeasible paths might be reported as feasible. However, this is safe, since less information about infeasible paths only gives a possibly less tight WCET estimate.

We have created an abstract analysis domain for the data representation in C [11]. This allows us to handle C features like structs, arrays, pointers and type casts. We do not perform our analysis directly on the C source code. Instead, it is applied on an intermediate code format, making our flow analysis more generic and less dependent on C source characteristics.

The infeasible path analyses presented in this paper are implemented in SWEET as a part of the abstract execution. The abstract execution is input data sensitive, as illustrated in Figure 1, allowing the user to constrain the possible input data values. The result of the abstract execution is passed as flow information, *flow facts* [8], to the subsequent calculation phase. Flow facts are a kind of constraints on the execution count.

4 Algorithms for Infeasible Paths

We now present our three algorithms for infeasible path detection. Since they are based on abstract execution, which is input-sensitive, the analyses are input-sensitive as well.

All three algoritms have a similar overall structure. They augment each analysis state with a *recorder* keeping track of nodes and path(s) taken during a particular analysis of a scope. Each algorithm resets the recorder of a scope when starting a new iteration of the scope. They also associate a *collector* to each scope, which accumulates information about nodes and paths during iterations of the scope. In the end, each collector is used to generate flow facts for its scope.

4.1 Detecting Infeasible Nodes

The first algorithm finds infeasible nodes, that is: basic blocks which are never visited in any execution of a certain scope. Since there is one scope per context, the resulting flow information becomes context sensitive. An infeasible node is therefore not necessarily the same as dead code, since the basic block potentially can be executed in another context.

The recorder object is a bit array with one bit per node in the scope. These bits are all reset to zero at each iteration of the scope, and the bit of a node is set to one at each abstract execution of the node. Thus, a value of zero, after an iteration, means "definitely not executed in this iteration" and one means "may have been executed in this iteration".

The collector object is a similar bit array. Its bits are all initialized to zero, and the end of each iteration the new value of the collector object is set to the bitwise or of its old value and the current value of the recorder object. At termination, if the collector holds a zero for a node, then it is surely never executed in that scope, and a corresponding "infeasible node flow fact" can be generated. An example is:

```
scope : <> : #BB82 = 0;
```

specifying that basic block `BB82` is not executed in any iteration of the scope `scope`.

4.2 Detecting Infeasible Pair of Nodes

The second algorithm finds infeasible pairs of basic blocks, i.e., blocks which are always excluding each other during the same iteration of a scope. This gives additional knowledge as compared to the first analysis, since there might be nodes which both can be executed during some iteration of a scope, but which never can be executed together. The limitation is that infeasible paths with more than two selections can be missed.

The recorder object for this algorithm is a path (list of nodes) taken during an iteration. To limit the number of recorded nodes, only nodes after conditional branches are recorded. At the entry of a scope or at a new loop iteration, the path is emptied. Whenever a conditional branch is taken, we remember the branch

by appending the corresponding node to end of the path. If both paths are taken, the analysis proceeds in two abstract states, one for each path.

The collector object is a triangular matrix which holds exclusion data. It is of size $N \times N$, where N is the number of possible branch outcomes (basic blocks) for the selections in the scope. The matrix can be triangular since the order of the elements in a pair is irrelevant. All elements in such a matrix are set to $\perp$ in the beginning of the analysis, which means that no information is available to start with. A recorder list RL is added to the collector matrix M when an abstract state has reached the end of a loop body or a function scope. The collector is updated as follows:

```
for each node n1 in RL do
  for each subsequent node n2 to n1 in RL do
     M[n1,n2] := 1
  for each alternative branch node n3 to n2 do
     if M[n1,n3] = ⊥ then
        M[n1,n3] := 0
     else
        M[n1,n3] := M[n1,n3] OR 0
```

For example, if the path **A-C** was taken in the example in Section 2 we would have updated M[**A**][**C**] to 1 and M[**A**][**D**] with 0.

When the analysis has finished, the resulting collector matrix is investigated. Matrix positions with $\perp$ mark pairs which have not been touched during the analysis. Some of them can never be executed together anyway due to the structure of the control graph, while the rest really are infeasible pairs. For the first type, generating flow facts will be superfluous. They could be identified using a reachability analysis. However, this is not included in the current implementation, so to avoid a large number of superfluous flow facts, no flow facts are currently generated for $\perp$ positions.

If the matrix positions holds a 0, it marks a node pair that we surely know excludes each other for any iteration of the scope, so for this pair an "excluding pair flow fact" can be generated, like:

```
scope : <> : (#BB33 + #BB57) < 2;
```

specifying that for any iteration of `scope` the basic blocks `BB33` and `BB57` are never executed together.

4.3 Detecting Infeasible Paths

The third algorithm finds sequences of nodes which are never executed together during the same iteration of a scope. The algorithm makes use of the fact that many infeasible paths can be efficiently represented by allowing them to share a common prefix (sub)path.

The recorder data object is now a tree where each tree node represents a path and has an associated

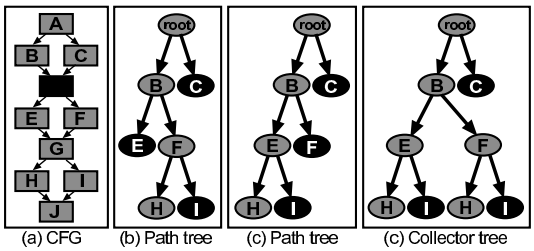


Figure 2. Example CFG and Path Trees

boolean specifying if the corresponding path is feasible or unfeasible. Similar to the recording in the second algorithm we only keep track of nodes taken after branches. However, the tree additionally keeps track of branch outcomes not taken.

Figure 2 gives an illustration of how the recorder tree works. Figure 2(a) gives a CFG with $2^3 = 8$ structurally possible execution paths. Figure 2(b) gives the tree resulting from an execution taking the path **A-B-D-F-G-H-J** through the CFG. In the recorded tree the paths **A-C**, **A-B-D-E** and **A-B-D-F-G-I** have been marked as infeasible. Note that we, for the sake of efficiency, do not record any join nodes in the tree. Similarly, Figure 2(c) gives the tree resulting from an execution path of **A-B-D-E-G-H-J**. Note that the path **A-C** actually represents $2^2 = 4$ number of paths through the CFG, since**A-C** is a prefix of all these paths.

For this algorithm the collector is the tree of paths obtained by merging all recorded trees for the scope. The basic idea of the collector is the same as in the first two algorithms, i.e., only keep infeasible path information which are true for all executions of the scope.

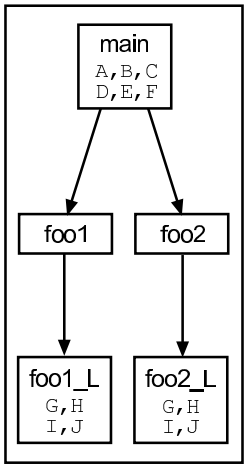
Figure 2(d) gives the collector tree resulting from merging the two trees in Figure 2(b) and Figure 2(c). Both trees has **A-C** as infeasible, and so does the collector tree. Since **A-B-D-E** is infeasible in Figure 2(b) but not in Figure 2(c), since it is part of the feasible **A-B-D-E-G-H-J** path, the collector tree cannot keep the path as infeasible. Instead, all paths (both infeasible and feasible) starting with the path **A-B-D-E** in Figure 2(c) are added to the collector tree. The infeasible path **A-B-D-F** in Figure 2(c) is extended similarly. The resulting collector tree in Figure 2(d) marks paths **A-C**, **A-B-D-F-G-I** and **A-B-D-E-G-I** as infeasible.

A collector tree CT is updated with a recorder tree RT as follows:

```
if RT is the first recorded tree reported
  CT := RT
else
  for each infeasible path i in CT do
    if i is prefix to a feasible path in RT
      mark path i as feasible in CT
      add all paths with prefix i in RT to CT
```

```
x=INPUT; // x=[0..100]
int main(void) {
   if (x<10) ...;      // A
   else ...;           // B
   if (x<5) ...;       // C
   else ...;           // D
   foo(x);
   if (x<0) ...;       // E
   else ...;           // F
   foo(x+50);
   return 1;
}
int foo(int y) {
   int i;
   for (i=0; i<10; i++) {
      if (y>=50) ...; // G
      else ...;       // H
      if (y<50) ...;  // I
      else ...;       // J
   }
}
```

(a) Example code



(b) Scopes

Figure 3. Code with several infeasible paths

After the analysis we create flow facts for the remaining infeasible paths in the collector tree. An example of such a flow fact is:

```
scope : <> : (#BB33 + #BB57 + #BB82) < 3;
```

specifying that basic blocks BB33, BB57, and BB82 are never executed together for each iteration of `scope`.

5 Example

The example code in Figure 3(a) contains infeasible paths of several types (we assume that neither `i`, `x` or `y` are changed in the excluded code). It will be used to illustrate the algorithms we propose in the paper. The program contains five scopes; `main`, `foo1`, `foo2` (the two calls to `foo`) and the corresponding loop scopes (`foo1_L` and `foo2_L`) in `foo`, as depicted in Figure 3(b). We can identify the following infeasible nodes, pairs and paths:

1. **Infeasible nodes:**
   - **E** is an infeasible node in `main`, **H** and **I** are infeasible nodes in `foo2_L` (limitations in input data).
2. **Infeasible pairs:**
   - **B-C**, **B-E**, and **D-E** are infeasible pairs in `main` (contradicting conditions).
   - **A-E** and **C-E** is an infeasible pair in `main` (limitations in input data).
   - **G-I** and **H-J** are infeasible pairs in `foo1_L` and `foo2_L` (contradicting conditions).
   - **H-I** is an infeasible pair in `foo2_L` (limitations in input data).
3. **Infeasible paths:**
   - **A-D-E**, **B-C-E**, **B-C-F**, and **B-D-E** are infeasible paths in `main` (contradicting conditions).
   - **B-D-E** is an infeasible path in `main` (limitations in input data).

We note that infeasibility can be expressed in several ways, e.g., the infeasible pair **B-C** and the infeasible paths **B-C-E** and **B-C-F** exclude the same paths.

6 Evaluation

Program	Description	#LC	#S	#L
adpcm	Adaptive pulse code modulation algorithm.	879	65	27
bs	Binary search for the array of 15 integer elements.	114	3	1
bsort100	Bubblesort program.	128	4	2
cnt	Counts non-negative numbers in a matrix.	267	10	4
compress	Compression using lzw.	508	22	11
cover	Program for testing many paths.	640	7	3
crc	Cyclic redundancy check computation on 40 bytes of data.	128	11	6
duff	Using "Duff's device" to copy 43 byte array.	86	5	2
edn	Finite Impulse Response (FIR) filter calculations.	285	21	2
expint	Series expansion for computing an exponential integral function	157	5	3
fdct	Fast Discrete Cosine Transform.	239	4	2
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.	219	52	30
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	3	1
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	276	4	2
insertsort	Insertion sort on a reversed array of size 10.	92	3	2
janne_complex	Nested loop program.	64	4	2
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	375	5	2
lcdnum	Read ten values, output half to LCD.	64	3	1
ludcmp	LU decomposition algorithm.	147	14	11
matmult	Matrix multiplication of two 20x20 matrices.	163	12	7
ndes	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	231	25	12
ns	Search in a multi-dimensional array.	535	6	4
nsichneu	Simulate an extended Petri net. Automatically generated code with more than 250 if-statements.	4253	2	2
qsort-exam	Linear equations by LU decomposition.	121	8	6
qurt	Root computation of quadratic equations.	166	16	3
select	A function to select the Nth largest number in a floating point array.	114	6	4
statemate	Automatically generated code.	1276	9	1

Table 1. Benchmark programs used

We have used programs from the Mälardalen WCET Benchmark to test our calculations. Table 1 gives some basic data about the programs (LC = lines of code), number of iteration scopes (#S), and number of (context-dependant) loops (#L). Table 2 shows the results of the different analyses. It shows the following information: Analysis time in seconds for abstract execution with loop bound analysis only (LB), number of found flow facts (#FF), and analysis time (Time) for each of the three algorithms (IN = infeasible nodes, EP = exclusive pairs, IP = infeasible paths). All measurements were performed on a 1.25 MHz PowerPC G4 processor, 1 Gb memory running Mac OS 10.4.6.

We see that we, with a small extra cost, can find infeasible nodes and paths for some of the benchmarks. It

Program	Time LB	Alg. 1 (IN) #FF	Alg. 1 (IN) Time	Alg. 2 (EP) #FF	Alg. 2 (EP) Time	Alg. 3 (IP) #FF	Alg. 3 (IP) Time
adpcm	19.14	37	19.86	44	19.22	24	20.04
bs	0.02	0	0.02	0	0.01	0	0.01
bsort100	0.95	3	0.95	0	0.95	0	0.96
cnt	0.21	1	0.22	0	0.21	0	0.23
compress	0.58	63	0.61	9	0.59	6	0.58
cover	0.71	114	1.65	1061	0.85	102	0.87
crc	2.13	18	2.36	6	2.16	4	2.24
duff	0.05	41	0.06	0	0.06	0	0.06
edn	1.22	0	1.23	0	1.23	0	1.29
expint	0.08	5	0.08	0	0.09	1	0.09
fdct	0.01	14	0.01	0	0.01	0	0.01
fft1	0.19	102	0.23	2	0.19	2	0.19
fibcall	0.02	0	0.02	0	0.02	0	0.02
fir	0.22	1	0.22	1	0.21	1	0.22
insertsort	0.13	0	0.13	0	0.13	0	0.12
janne_complex	0.02	1	0.03	4	0.03	0	0.02
jfdctint	0.03	0	0.03	0	0.03	0	0.03
lcdnum	0.01	41	0.02	6	0.02	6	0.02
ludcmp	1.88	3	1.88	1	1.89	1	1.88
matmult	2.76	0	2.79	0	2.84	0	2.99
ndes	8.02	11	9.39	3	8.10	1	8.13
ns	1.00	1	1.01	0	1.01	0	1.05
nsichneu	12.88	126	13.16	78150	1288.76	623	19.15
qsort-exam	0.18	1	0.19	11	0.18	6	0.19
qurt	0.08	27	0.11	7	0.08	5	0.08
select	0.21	2	0.20	8	0.19	14	0.19
statemate	0.14	256	0.15	5	0.13	32	0.13

Table 2. Analysis results

should be noted that these results are based on single-path analysis, i.e., using a single input that leads to a single execution path. We expect more infeasible nodes and paths to be found when we analyse the programs with inputs that leads to multi-path analyses.

7 Conclusions and Future Work

We do think that our results are promising, but they are still somewhat preliminary: the benchmarks used so far are limited to single-path programs, and we only count the number of generated flow facts for infeasible paths. The next step is to extend the evaluation to a larger set of benchmarks, using multi-path analysis, and to also investigate the effect of the derived infeasible path information on the WCET estimate. In particular, we want to try out the algorithms on industrial real-time codes.

We also want to investigate tradeoffs between analysis time and WCET estimate precision. One possibility would be to generate flow information for individual iterations of a scope. This could give tighter WCET estimates, at the expense of longer analysis times. Another possibility is to generate non-context-sensitive flow facts, valid for all different call-sites of a particular function or loop. This will, in general, give less precise WCET estimates, but for a lower analysis cost.

References

[1] H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Proc. 19th IEEE International Performance, Computing, and Communications Conference (IPCCC2000)*. IEEE, February 2000.

[2] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop of Real-Time Systems*, pages 102–107, June 1996.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation; Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer-Verlag, 2002.

[4] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In Reinhard Wilhelm, editor, *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 40–43, Palma de Mallorca, July 2005.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[6] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

[7] O. Eriksson. Evaluation of Static Time Analysis for CC Systems. Master's thesis, Mälardalen University, August 2005.

[8] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[9] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 19–22, July 2005.

[10] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

[11] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, February 2005.

[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)*, June 1999.

[13] Apostolos A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proc. 3rd International Conference on Real-Time Computing Systems and Applications (RTCSA'96)*. IEEE, IEEE Computer Society Press, 1996.

[14] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pages 31–40, June 1998.

[15] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.

[16] Mälardalen University WCET project homepage, 2006. `www.mrtc.mdh.se/projects/wcet`.

[17] Daniel Sehlberg. Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, Västerås, Sweden, October 2005.

Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup *

Raimund Kirner, Markus Grössing, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien, Austria
raimund@vmars.tuwien.ac.at

Abstract

Trigonometric functions are often needed in embedded real-time software. To fulfill concrete resource demands, different implementation strategies of trigonometric functions are possible.

In this paper we analyze the resource demands of iterative calculations compared to other implementation strategies, using the trigonometric functions as a case study. By analyzing the worst-case execution time (WCET) of the different calculation techniques of trigonometric functions we got the surprising result that the WCET of iterative calculations is quite competitive to alternative calculation techniques, while their economics on memory demand is far superior. Finally, a discussion of the general applicability of the obtained results is given as a design guide for embedded software.

1 Introduction

For real-time systems in safety-critical environments it is indispensable to design the temporal behavior of the system based on knowledge of the worst-case execution time (WCET) of the real-time tasks. A general discussion on research directions in the area of WCET analysis can be found in [9].

Besides analyzing the timing behavior of programs we also look at software design techniques that proactively simplify the analysis of the WCET. We have described a general paradigm, which we call WCET-oriented programming [10]. The basic idea of WCET-oriented programming can be summarized as the search for algorithms whose execution-time variability is small, for example, by avoiding input-data dependent control flow decisions whenever possible.

In this paper we study the characteristics of iteration-based computation techniques, we address interesting questions like whether these algorithms are suitable for real-time computing. For example, it is a common belief that iteration-based computation is critical, because a) long execution times due to high number of needed iterations and b) the problem of finding a precise upper iteration bound.

In the here-presented case study, we look at the behavior of trigonometric functions. The contribution of this paper is to connect the known properties of trigonometric functions to implementation techniques of real-time software and to provide an analysis of relevant characteristics like computation time and memory demands. Besides the interesting results obtained from our analysis, we also describe the application of WCET analysis to embedded software with floating point emulation.

2 Related Work

The work in this paper focuses on the properties of time-memory tradeoffs for real-time software. For example, one might design algorithms with shorter execution time by using more memory.

Sorting examples are *sorting by counting*, where a second array is used to sort elements with known relative positions based on their key in linear time, and *Radix Sort* [6]. Alternatively, *lookup tables* (LUT) can be used to reduce online calculations by deploying precalculated values. As an example for the use of lookup tables, see [7]. Time-space tradeoffs on dictionary attacks to break passwords are presented in [8]. Three further examples of applying time-memory tradeoffs are described in [11].

*This work has been partially supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

3 Trigonometric Functions

For our study of different computation techniques we focus on trigonometric functions because they are heavily used in many scientific disciplines. *Sine*, *cosine* and *tangent* as well as their inverse functions play important roles not only in surveying, navigation, or scientific mathematics, but also in many other fields like acoustics, astronomy, computer graphics, electrical engineering and electronics, mechanical engineering, optics, etc.

First of all, it is important to keep in mind that the requirements on trigonometric functions are quite different depending on the application domain. For example, an application domain where performance is typically more important than precision is 3D computer graphics. The cosine is a fundamental operation in 3D rendering techniques like various shading methods, ray tracing, etc. [12]. Those rendering techniques have to use the trigonometric functions excessively often. Therefore, effective approximation techniques are very important to gain performance, while high precision is not a first-order requirement.

However, we are focusing more on the use of trigonometric functions in the domain of embedded real-time systems. They are used in mechanical applications, e.g., to determine distances in automation systems, or for controlling the movement of a robotic arm. Other important real-time applications are multimedia systems, where they are used to compute Fourier transforms (e.g., for audio processing) or discrete cosine transforms (for graphics) are performed. Further interesting application fields are applications that use ultrasound, optical devices, or statistical computations.

For the application of trigonometric functions in embedded real-time control systems, typically both, the numerical precision and the resource demands are relevant. For the following discussions of different calculation techniques we concentrate on the *cosine* function, since the other trigonometric functions are closely related respectively can be derived from it. We also discuss the maximum error for each calculation technique, which is needed for the comparison of iterative calculation and table-lookup in Section 4.

3.1 Iterative Approximation (Taylor Series)

In common implementations of trigonometric functions the Taylor series is used to approximate sine, cosine and tangent. There also exist other iterative algorithms like CORDIC [1], which are slower than Taylor series but easier to implement in hardware as it does not need multiplication operations. In this paper we focus on the Taylor series because we are interested in implementations in software. Let us consider the power series implementation of cosine (Equation 1): to reach full *double* precision (as defined in [3]) a Taylor polynomial of degree 14 is needed. As only the coefficients of even powers are significant to calculate the cosine function, only seven coefficients are needed. We call this class of cosine implementation techniques CTAYLOR.

$$\cos(x) = \sum_{n=0}^{\infty}(-1)^n \cdot \frac{x^{2n}}{(2n)!} \approx \sum_{n=0}^{7}(-1)^n \cdot \frac{x^{2n}}{(2n)!} \quad (1)$$

The constant coefficients do not need to be calculated at runtime everytime the function is called. Instead, they can be stored as static constants.

The accuracy of the power series decreases as the distance of the argument from the center grows. Therefore for trigonometric functions this distance is limited to $\pi/4$. As the center of sine, cosine, and tangent approximation is chosen to be zero the actual evaluation interval of these functions is $[-\pi/4; \pi/4]$. To evaluate arguments outside this interval an argument reduction needs to be performed [7].

To estimate the maximum error of a Taylor series implementation we need to consider the error at $\pi/4$, where the distance to the center of the power series is maximal.

The maximum error of Taylor series with n iterations is given by the remainder term R_{n+1} in Equation 2.

$$R_{n+1} = \frac{1}{n!}\int_0^x (x-t)^n \cdot \cos^{(n+1)}(t)dt =$$
$$= \cos(x) - \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} \cdots + (-1)^n \cdot \frac{x^{2n}}{2n!}\right) \quad (2)$$

3.2 Approximation using Lookup Tables

Lookup tables are commonly used to replace runtime calculations with simpler lookup operations. Retrieving an array value from memory is usually much faster than making an expensive computation.

In the following we will take a look on three different implementations of lookup tables:

- Fast and simple lookup tables (*FLUT*)
- Equidistantly interpolated lookup tables (*EDILUT*), and
- Lookup table with interpolation with smart placement of interpolation points (*SMILUT*)

3.2.1 Fast and Simple Table Lookup

The fast and simple lookup table (FLUT) is nothing more than a data array that stores pre-calculated function values of the function in it. The places where these values are taken are equidistant, so the array index fitting to a given argument can easily be computed. Each value in the array covers an interval of arguments. The biggest error occurs, where the function has its greatest gradient (see Equation 3. For example, a cosine lookup table in the interval $[0; \pi/2]$ has its greatest gradient at $\pi/2$.

$$E_{max}(n) = \cos\left(\frac{\pi}{2} - \frac{\pi}{4(n-1)}\right) = \sin\left(\frac{\pi}{4(n-1)}\right) \quad (3)$$

The advantage of FLUT is that it is easy to implement and the estimation of the timing behavior is simple. The performance according to speed is very good but accuracy requirements should not be too high. To enhance accuracy or to reduce the table size if a particular level of accuracy is given other methods like EDILUT and SMILUT can be used.

3.2.2 Table Lookup with Equidistant Interpolation

An equidistant interpolated lookup table (EDILUT) reaches significantly higher accuracy compared to a FLUT of the same size. The price to pay is a little more arithmetics and so longer execution time.

The entries of an EDILUT are the function values of equidistantly distributed places of the input interval. In the case of a *cosine* EDILUT the input interval is $[0; \pi/2]$. In a cosine calculation, first the two interpolation points next to the given argument are determined. Then a straight line through these two points is calculated and the argument is set into this straight interpolation line. With this method accuracy can be increased significantly.

The maximum error of EDILUT occurs not on the place with the greatest gradient, like it was the case for FLUT, but on the place with the greatest curvature. For our *cosine* function this is the case near the origin, so we expect the greatest error to occur in the first interpolation interval. The error of the interpolation in the first interval can be calculated by subtracting the linear interpolation between the first two interpolation points from the function. By deriving this function and setting to zero, the exact place of the maximum error is retrieved. Applying this value to the error function gives the maximum absolute error (Equation 4) for a concrete lookup table size n of EDILUT.

$$E_{max}(n) = \cos\left(\sin^{-1}(k)\right) - k \cdot \sin^{-1}(k) - 1,$$
$$k = -\frac{1 - \cos\frac{\pi}{2(n-1)}}{\frac{\pi}{2(n-1)}} \quad (4)$$

3.2.3 Table Lookup with Smart Interpolation

A smart interpolated lookup table (SMILUT) is a further improvement of EDILUT. In a SMILUT the interpolation points are not equidistantly distributed in the input interval but in a smarter way. The function for mapping the input interval into the range of array indices should be rather simple. We map the input interval to the indices using the squareroot function. The result is an improvement of accuracy.

With this placement we achieve that the maximum error does not occur within the first interpolation interval but rather in the middle of the overall input interval.

As the squareroot function might be too expensive to compute, we considered an alternative implementation for finding the correct interpolation interval, namely using binary search.

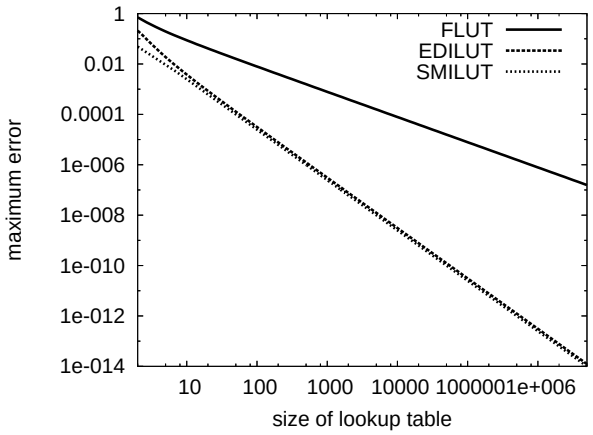


Figure 1. Maximum Absolute Error of FLUT, EDILUT, and SMILUT

To complete the discussion about LUT-based solutions, a comparison of the accuracy in dependence of the size of the lookup array is given in Figure 1 [1].

[1] Note that due to range limitations in the numerical calculation, the size values above 2000 are extrapolated values to show the tendency of the graph.

4 Comparison of Iterative vs. LUT-based Techniques

A comparison of different iteration numbers of Taylor series implementations to the three lookup table approaches is depicted in Figure 2. It is shown how many entries a particular type of lookup table needs to exceed the accuracy of different Taylor series implementations.

As the maximum error of FLUT and EDILUT can be calculated analytically, these two variants can be easily compared to the Taylor series. For the comparison of SMILUT a simple tool was developed to experimentally determine the required size of the SMILUT to reach the accuracy of the different Taylor series implementations. This tool determines the maximum error of a SMILUT for a given array size. If the error is too big the array size is increased. The program terminates when the accuracy of the SMILUT exceeds a given limit, e.g., the accuracy of a particular Taylor implementation.

As shown in Figure 2 the accuracy of FLUT is much worse than the accuracy of the other LUT implementations. SMILUT performs slightly better than EDILUT. One can see that the size of lookup tables of any type grows exponentially with the number of Taylor iterations. Thus, if high accuracy requirements need to be met, the use of lookup tables may not be feasible or sensible to approximate trigonometric functions - the memory consumption of these algorithms is too high.

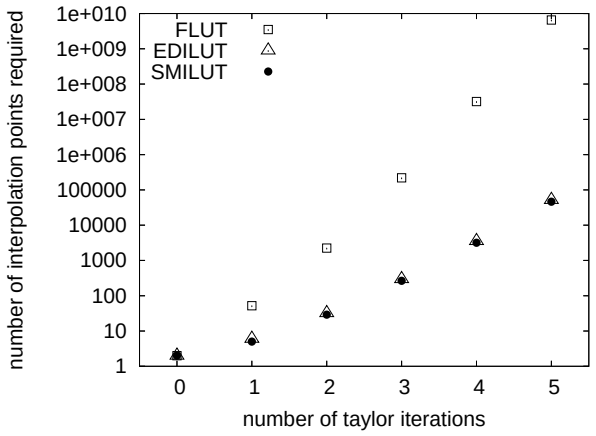


Figure 2. Necessary LUT Size to Match the Accuracy of Taylor Series

5 Experimental Evaluation

In Section 3 we described the theoretical properties of different calculation techniques of trigonometric functions. Lets now look at the different calculation techniques from a practical point of view. Especially interesting for the use of trigonometric functions in embedded real-time systems are their resource demands. Therefore, we analyzed their memory footprints in data and code memory, and calculated an upper bound of their worst-case execution time (WCET).

5.1 Studied Algorithms

To analyze the properties of the different computation techniques discussed in Section 3 on a concrete computer platform, we implemented several variants of the cosine function. We implemented the cosine function for the *double* data type of ANSI C (which is typically the 64-bit IEEE floating point format [3]).

Two iterative cosine variants belonging to CTAYLOR were implemented, one which is a straight forward implementation of the Taylor-formula and one with precalculation of the coefficients of the Taylor-terms.

On the other side the three LUT-based variants of Section 3.2 have been implemented: FLUT the straight forward method, EDILUT which uses linear interpolation and SMILUT, an implementation using binary search to find the correct interpolation point within the LUT. Compared to EDILUT it is highly performance oriented and uses more precomputed results, requiring three LUTs.

Some characteristic parameters of the different cosine implementations are given in Table 1. The column *#BB* denotes the number of *basic blocks* of the generated object code. The *DataMem* columns give the required number of bytes to store the intermediate data and the LUT. It is given first in parametric form as a function of the LUT size N, and second for the concrete case $N = 1000$. The column *CodeMem* denotes the net code size, i.e., without counting the standard library functions which are linked by the compiler. The byte values are given for the Infineon C167 processor, a 16bit architecture.

5.2 WCET Analysis

In the following we describe how we derived the WCET of the different cosine implementation techniques. Our WCET analysis tool *calc_wcet_167* [2] uses static timing analysis to calculate an upper bound of a task's WCET. The target architecture for the tool is the processor C167 from Infineon, for which the GCC compiler was ported by the company *HighTec EDV*

[2] `http://www.wcet.at/tools.html`

Function name	*#BB*	*DataMem* (parametric) [bytes]	*DataMem* (N=1000) [bytes]	*CodeMem* [bytes]
CTAYLOR	22	28	n.a.	720
CTAYLOR_tab	19	90	n.a.	598
FLUT	13	10+N·8	8 010	456
EDILUT	15	34+N·8	8 034	902
SMILUT	23	30+(N+1)·24	24 054	536

Table 1. Implemented Calculation Variants of the Cosine Function

Systeme GmbH [3]. The integration of optimizing compilation into the WCET analysis is described in [5]. The development and verification of the timing model for the Infineon C167 is documented in [2]. Because the Infineon C167 processor has a relatively simple architecture, the overestimation of the calculated WCET bound of our tool is tight, maximal 5%, but typically less than 2%, provided the control flow is precisely modelled by flow constraints [2].

The cosine implementations were written in WCETC, based on a subset of ANSI C but providing additional features to annotate the source code with flow information to guide the WCET analysis tool [4].

The WCET analysis of the cosine implementations itself did not require anything special to mention. However, the overall WCET analysis was not easy because the Infineon C167 processor does not have a floating point engine in hardware. For such architectures the compiler links extra program code that emulates the floating point computations in software (`libsgnu.a` provided by *HighTec*). To perform the WCET analysis we disassembled the object code of the library and annotated it at assembly code level with flow information.

The final WCET analysis results of the different cosine implementations are given in Table 2. Besides the properties of the concrete implementations, these values are generally rather high because we assumed a slow hardware configuration with slow external memory. The first of the WCET columns shows the WCET in a parametric form. This is only relevant for the iterative implementations, where *iter* is the number of loop iterations used to iteratively refine the result based on the Taylor series of the cosine function. The other four WCET columns show the WCET bound of the iterative algorithms for different iteration counts.

5.3 Discussion

The results of the general analysis of the maximum absolute error for CTAYLOR and of the maximum absolute error of FLUT, EDILUT, and SMILUT together with the precision relationsship between CTAYLOR and LUT-based methods (Figure 2) can be combined with the results from the concrete implementation to reason about the pros and cons of the different computation paradigms.

To demonstrate how this can be done, lets assume that for a concrete project one needs a cosine function providing a maximal error of less than $2.5 \cdot 10^{-8}$. Evaluating Equation 2 at $\pi/4$ it follows that one would need 4 iterations ($\equiv$ 5 Taylor terms) with the CTAYLOR methods. To replace the CTAYLOR method later by an adequate LUT-based method one could deduce from Figure 2 the required LUT size to match at least the same precision. For example, to obtain the same quality with EDILUT or SMILUT, one has to choose an LUT size of $N > 3000$!

From Table 1 we can see that in this case the additional memory demand for the LUT-based methods is significant. The FLUT method, though it is relatively fast, is completely out of choice as it would require an LUT size of $N > 10^5$. If performance really is the most important issue, then according to Table 2 one has to use the SMILUT implementation. But surprisingly, the CTAYLOR methods are not that bad regarding the WCET compared to SMILUT. As a rough indicator using our example, one would need only `90` bytes data memory when using CTAYLOR_tab compared to the more than `72kB` when using SMILUT. The code size is almost the same between these two implementations.

Another benefit of the CTAYLOR methods is their *anytime* characteristic. In case a CTAYLOR method gets interrupted, there is still some accuracy of the result available, for example, to move a robot arm at least in the intended direction, hoping that the control will be refined in the next round by a more accurate result. Depending on the application, this can be an advantage of iteration-based CTAYLOR methods compared to the LUT-based methods.

[3] `http://www.hightec-rt.com`

Function name	*WCET* [cycles] (parametric)	(iter=1)	(iter=3)	(iter=4)	(iter=7)
CTAYLOR	23 140 + iter·79 500	102 640	261 640	341 140	579 640
CTAYLOR_tab	23 380 + iter·49 200	72 580	170 980	220 180	367 780
FLUT	136 840	n.a.	n.a.	n.a.	n.a.
EDILUT	276 640	n.a.	n.a.	n.a.	n.a.
SMILUT	120 540	n.a.	n.a.	n.a.	n.a.

Table 2. Calculated WCET of the Cosine Functions (Target Processor: Infineon C167)

Generality of the Results

In our concrete case study of the cosine function it has been shown that the WCET of the iterative calculation is quite competitive to table-lookup while the economics on memory demand is far superior.

Iterative calculations generally provide the same advantages as long as they have a relatively compact calculation step within each iteration and the termination speed of the iterative calculation is reasonable.

As a further example, the *Newton method* to solve equations numerically tends to provide such a behavior, provided that the start value is already within the local convergence interval of the solution. There are many instantiations of the Newton method in practice, e.g., the *Heron method* to calculate the square root of a number.

6 Summary and Conclusion

Motivated by our general effort to study the suitability of different algorithms for real-time computing, we looked at different calculation techniques of trigonometric functions, because of their use in a wide range of technical applications.

One of the central conclusions is that whenever memory is highly constrained, iteration-based methods are very useful, because they tend to demand much less memory while still providing reasonable accuracy of results. And quite important, the performance overhead of iteration-based methods is not that high, even in our case study where we calculated the WCET for the C167, a processor that emulates floating point arithmetics in software. Further, the WCET analysis itself was an interesting experience, as we had to analyze routines of the floating-point emulation library by disassembling the object code.

In general, in embedded real-time systems, where size of memory is typically restricted, iterative algorithms can be a memory-efficient calculation technique without significant performance costs compared to LUT-based methods, as long as a reasonable termination speed of the iterative calculation is ensured.

References

[1] R. Andraka. A survey of cordic algorithms for fpga based computers. In *Proc. ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays*, pages 191–200, 1998.

[2] P. Atanassov. *Experimental Assessment of Worst-Case Program Execution Times.* PhD thesis, Technische Universität Wien, Vienna, May 2003.

[3] IEEE. *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.* IEEE, New York, 1987. Reprinted in SIGPLAN Notices 22,2,9-25.

[4] R. Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[5] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003.

[6] D. E. Knuth. *The Art of Computer Programming - Sorting and Searching*, volume 3. Addison Wesly, New York, USA, 2nd edition, 1998. ISBN 0-201-89685-0.

[7] J. N. Lygouras. Memory reduction in look-up tables for fast symmetric function generators. *IEEE Transactions on Instrumentation and Measurement*, 48(6):1254–1258, Dec. 1999.

[8] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. 12th ACM conference on Computer and Communications Security*, pages 364–372, New York, NY, USA, 2005. ACM Press.

[9] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[10] P. Puschner and R. Kirner. Avoiding timing problems in real-time software. In *Proc. IEEE Computer Society's Workshop on Software Technologies for Future Embedded Systems*, May 2003.

[11] M. Stamp. Once upon a time-memory tradeoff. Technical report, San José State University, San José, California, USA, July 2003.

[12] A. Watt. *3D Computer Graphics.* Addison Wesley, 3rd edition, Dec. 1999. ISBN: 0201398559.

History-based Schemes and Implicit Path Enumeration

Claire Burguière and Christine Rochange
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
31062 Toulouse cedex 9, France
{burguier,rochange}@irit.fr

Abstract

The Implicit Path Enumeration Technique *is often used to compute the WCET of control-intensive programs. This method does not consider execution paths as ordered sequences of basic blocks but instead as sets of basic blocks with their respective execution counts. This way of describing an execution path is adequate to compute its execution time, provided that safe individual WCETs for the blocks are known. Implicit path enumeration has also been used to analyze hardware schemes like instructions caches or branch predictors the behavior of which depends on the execution history. However, implicit paths do not completely capture the execution history since they do not express the order in which the basic blocks are executed. Then the estimated longest path might not be feasible and the estimated WCET might be overly pessimistic. This problem has been raised for cache analysis. In this paper, we show that it arises more acutely for branch prediction and we propose a solution to tighten the estimation of the misprediction counts.*

1 Introduction

The difficulty of evaluating the Worst-Case Execution Time of a real-time application comes from the – generally – huge number of possible paths that makes it intractable to analyze each of them individually. The single-path programming paradigm [9] would noticeably simplify the WCET computation but it has a cost in terms of performance that might not be acceptable. This is the reason why much research effort has been put on developing WCET evaluation approaches based on static analysis [10]. These approaches factorize the efforts by building up the WCET of the complete program from the individual WCETs of basic blocks. The Implicit Path Enumeration Technique [13], also known as *IPET*, is a very popular method for WCET calculation. It expresses the search of the WCET as an Integer Linear Programming problem where the program execution time is to be maximized under some constraints on the execution counts of the basic blocks. With this technique, an execution path is defined by the set of the executed blocks with their respective execution counts but the order in which they are executed is not expressed.

More and more complex processors are used in real-time embedded systems and it is a real challenge to take into account all of their advanced features in WCET analysis. In particular, some mechanisms have a behaviour that depends on the execution history which is difficult to capture by static analysis. These mechanisms include cache memories and dynamic branch predictors. In this paper, we consider bimodal branch prediction as an example of such schemes.

Various methods to take branch prediction into account have been proposed in the litterature. As explained below, we focus on the approach by Li *et al.* [12] that we have later extended [5] to take into account 2-bit prediction counters. In this paper, we show that both models can lead to over-estimated WCET because the worst-case number of mispredictions computed by IPET would correspond to an infeasible execution path. We show how they should be revised to only reflect feasible behaviours of the branch prediction scheme. Experimental results show that the revised model tighten the estimated WCET.

The paper is organized as follows. In Section 2, we illustrate the differences between implicit and explicit execution paths by an example. Section 3 gives an overview of dynamic branch prediction and lists previous work on branch prediction modeling for WCET analysis. We show how misprediction counts can be over-estimated in Section 4 and we propose an extended model to tighten the estimated WCET in Section 5. Section 6 concludes the paper.

2 Implicit vs. explicit execution paths

Figure 1 gives an example code that will be used throughout this paper and Figure 2 shows the corresponding CFG.

```
#define M 4
#define N 6
int main() {
  int i, j;
  int mat[M][N];
  for (i=0; i<M; i++){
     mat[i][0]=1;
     for (j=0; j<N; j++)
        mat[i][j]=i+j;
  }
}
```

Figure 1. Example code.

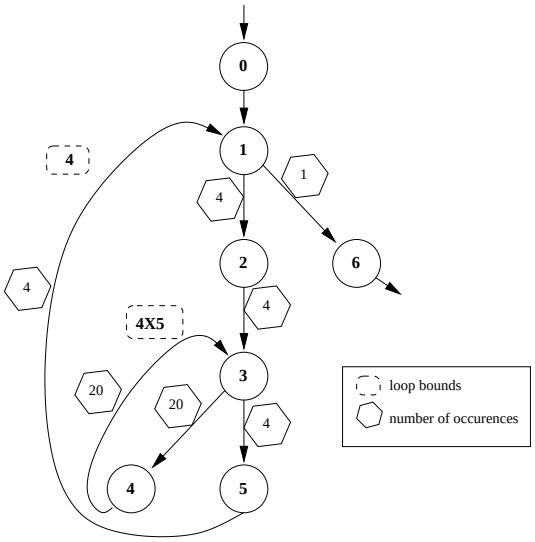


Figure 2. Example CFG.

Explicit paths. An *explicit* execution path is an ordered list of the executed basic blocks. In our example, the only possible execution path is defined by the sequence of blocks:

$$\Big(\mathtt{b1}-\mathtt{b2}-(\mathtt{b3}-\mathtt{b4})_{\times 5}-\mathtt{b3}-\mathtt{b5}\Big)_{\times 4}-\mathtt{b1}-\mathtt{b6}$$

Path-based WCET analysis [11][16] explores explicit paths but this might be costly because, as said before, a program may have many possible (explicit) paths. Some static WCET analysis methods simplify the path exploration while still considering explicit paths. For example, the Extended Timing Schema [17][15] works on the Syntax Tree.

Implicit paths. The IPET method [13] considers *implicit* paths. An implicit path is defined by the list of its basic blocks and of their execution counts. The implicit path corresponding to the explicit path given above is:

$$\big(\mathtt{b1}_{\times 5}, \mathtt{b2}_{\times 4}, \mathtt{b3}_{\times 24}, \mathtt{b4}_{\times 20}, \mathtt{b5}_{\times 4}, \mathtt{b6}_{\times 1}\big)$$

An implicit path defines many possible explicit paths but, in general, most of them are infeasible. For example, the implicit path given above could be expanded as below, where the inner loop is executed three times with a single iteration and once with 17 iterations, which is not consistent with the program semantics.

$$(\mathtt{b1}-\mathtt{b2}-\mathtt{b3}-\mathtt{b4}-\mathtt{b3}-\mathtt{b5})_{\times 3}-\mathtt{b1}-\mathtt{b2}-\\ -(\mathtt{b3}-\mathtt{b4})_{\times 17}-\mathtt{b3}-\mathtt{b5}-\mathtt{b1}-\mathtt{b6}$$

In the IPET method, the execution time of an implicit path is computed by adding the individual execution times of the basic blocks weighted by their execution counts. The WCET is obtained by maximizing the total execution time under some contraints that link the execution counts of the nodes and edges of the CFG: structural constraints directly express the CFG structure and flow constraints express loop bounds and infeasible paths.

As long as implicit paths are only used to compute a global execution time by summing individual times, there is no need to provide further information about the program semantics. However, when some mechanisms based on the execution history have to be modeled within the same framework, the *implicit* expression of execution paths might not be sufficient. This problem has been raised in the case of instruction cache analysis [14]. In this paper, our purpose is to show that it can also arise when modeling branch prediction and that it is a bit more complex in this case. However, we will provide a solution to get round it.

3 Branch prediction and WCET estimation

3.1 Bimodal branch prediction

Branch prediction enhances the pipeline performance by allowing the speculative fetching of instructions along the predicted path after a conditional branch has been encountered and until it is resolved. If the branch was mispredicted, the pipeline is flushed and the other path is fetched and executed. In the hardware *bimodal* branch predictor [18], the branch direction is predicted from a 2-bit saturating counter stored in the *Branch History Table* (which is indexed by the branch PC). If the branch is predicted as taken (counter equal to **11** or **10**), the target address is read in the *Branch Target Buffer*, otherwise the instruction fetch proceeds sequentially. When the branch is later computed, the prediction counter is updated as shown in Figure 3.

3.2 Modeling branch prediction for WCET estimation

3.2.1 Background

Modeling bimodal dynamic branch predictors for WCET analysis has been the purpose of several papers these last

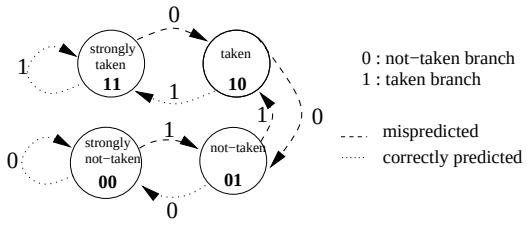


Figure 3. Bimodal branch prediction.

years. Some of the proposed techniques are *decoupled* from the pipeline analysis. Colin and Puaut [8] use static simulation to determine whether the prediction counter associated to a branch might be corrupted by another branch. Then they combine these results with an analysis of the behaviour of the 2-bit counters related to algorithmic structures to calculate bounds on the misprediction counts. Other works assume that branch aliasing can be prevented and refine the analysis of branching patterns related to algorithmic structures [1][6]. Once misprediction counts have been determined, the estimated execution time is augmented by the corresponding misprediction penalties.

To take into account tigher *per-branch* misprediction penalties, branch prediction modeling can also be *integrated* to the WCET computation with IPET [2][4]. Li *et al.* go further by completely modeling the behaviour of the branch prediction scheme within the IPET model [12]. They take conflicts in the Branch History Table into account but they only consider 1-bit prediction counters. In [5] we argue for techniques to prevent aliasing and we extend their model by considering 2-bit counters. Our discussion here is based on this last work.

Modeling branch prediction as part of WCET computation has several advantages: (1) any kind of loop can be analyzed, even if it is not well structured (*e.g.* several exit points); (2) the analysis of branches implementing conditional structures does not require any particular effort; (3) per-branch misprediction penalties can be specified.

3.2.2 Baseline model

The estimation of misprediction counts is combined to WCET computation by IPET by the way of additional constraints that: (a) express the way the prediction counters evolve; and (b) link the evolution of the prediction counters to the execution counts of the blocks and edges in the CFG. In this section, we give a simplified overview of the model. The variables used to evaluate the WCET by IPET are:

x_i	execution count of block i
$x_{i\xrightarrow{d}}$	execution count of the edge leaving block i when the branch direction is d $x_i = x_{i\xrightarrow{0}} + x_{i\xrightarrow{1}}$

The constraints added to model a bimodal branch predictor (without aliasing in the Branch History Table) use some additional variables (execution counts) presented in Figure 4.

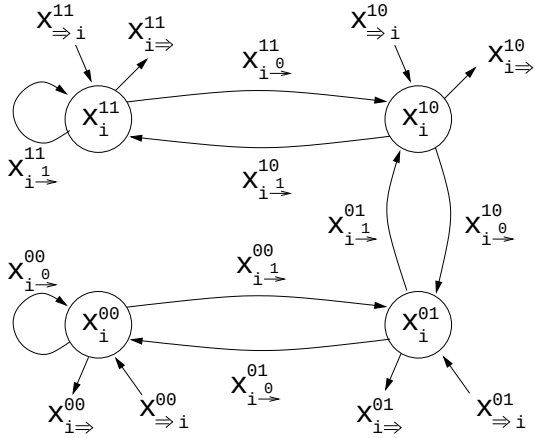


Figure 4. Variables used to model branch prediction for block i

The set of possible states for a 2-bit branch prediction counter is denoted as $\mathcal{C} = \{00, 01, 10, 11\}$ and the set of possible directions d after a branch is denoted as $\mathcal{D} = \{0, 1\}$. The constraints that model the way the prediction for the branch of block i evolves are:

$$\left.\begin{aligned} x_i^{00} &= x_{i\xrightarrow{0}}^{00} + x_{i\xrightarrow{0}}^{01} + x_{\Rightarrow i}^{00} \\ x_i^{01} &= x_{i\xrightarrow{0}}^{10} + x_{i\xrightarrow{1}}^{00} + x_{\Rightarrow i}^{01} \\ x_i^{10} &= x_{i\xrightarrow{0}}^{11} + x_{i\xrightarrow{1}}^{01} + x_{\Rightarrow i}^{10} \\ x_i^{11} &= x_{i\xrightarrow{1}}^{11} + x_{i\xrightarrow{1}}^{10} + x_{\Rightarrow i}^{11} \end{aligned}\right\} \quad (1)$$

$$\forall c \in \mathcal{C}, x_i^c = x_{i\xrightarrow{0}}^c + x_{i\xrightarrow{1}}^c + x_{i\Rightarrow}^c \quad (2)$$

The variables related to branch prediction are linked to the execution counts of basic blocks and edges by the following constraints:

$$x_i = \sum_c x_i^c \qquad \forall d \in \mathcal{D}, x_{i\xrightarrow{d}} = \sum_c x_{i\xrightarrow{d}}^c + \sum_c x_{i\Rightarrow}^c \quad (3)$$

For the initial and final state of the branch counter of block i, we can write:

$$\sum_c x_{\Rightarrow i}^c = 1 \qquad \sum_c x_{i\Rightarrow}^c = 1 \quad (4)$$

Finally, mispredictions counts are derived from:

$$m_i = x_{i\xrightarrow{1}}^{00} + x_{i\xrightarrow{1}}^{01} + x_{i\xrightarrow{0}}^{10} + x_{i\xrightarrow{0}}^{11} \quad (5)$$

4 Branch prediction modeling and implicit path enumeration

4.1 Example code

We have analyzed the branch predictor behavior for our example code using the model described in the previous section. Figure 5 shows the results we obtained. The branch at the end of block 3 controls the inner loop that iterates 5 times (then the branch is executed 6 times for each execution of the loop: it is *not taken* 5 times and *taken* once). The inner loop is repeated 4 times: block 3 is then executed 24 times on the global execution path (20 times as *not taken* and 4 times as *taken*). The numbers given in the dotted and dashed hexagons stand for the correct and erroneous branch prediction counts.

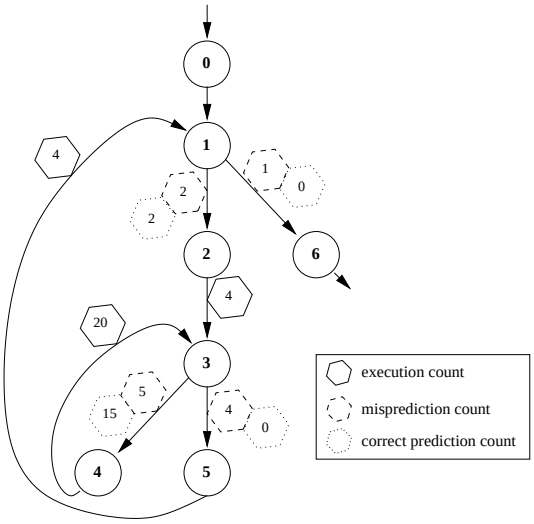
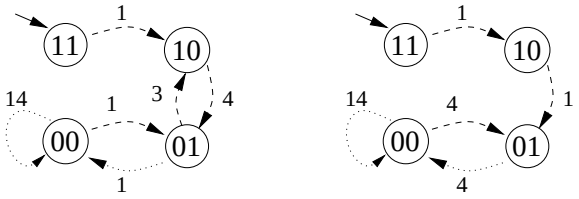


Figure 5. Results for the example code

In [1], the worst-case number of mispredictions for a branch that controls a repeated loop is bounded. For a loop that iterates N times ($N \geq 3$) and is repeated M times, the worst-case misprediction count is $(M + 2)$: during the first execution of the loop, the prediction counter reaches the `00` state after 3 iterations at most (considering any possible starting state) and the branch is mispredicted at most twice. At the end of every execution of the loop, the counter is incremented from `00` to `01` and the branch is mispredicted (this makes M mispredictions). For the next execution, the counter is decremented to `00` and the branch is well predicted.

According to these results, the worst-case misprediction count for the inner loop of our example should be 6. But the result obtained with the ILP model is 9, as shown in Figure 5. A closer look to these results show that they correspond to a behaviour of the prediction counter as the one



(a) erroneous **(b) correct**

Figure 6. Detailed results for counter state transitions

shown in Figure 6 (a). This behaviour would be obtained if the branching pattern was:

$$\Big((\texttt{NT} - \texttt{NT} - \texttt{T}) - (\texttt{NT} - \texttt{T})_{\times 2} - (\texttt{NT}_{\times 16} - \texttt{T})\Big)$$

where `NT` stands for "*not taken*" and `T` for "*taken*". This pattern defines a path where the inner loop executes once with 2 iterations, then twice with one iteration, and finally once with 16 iterations: this path is inconsistent with the program semantics.

The only possible *explicit* path for this program has the same *implicit* description but has the following branching pattern:

$$(\texttt{NT} - \texttt{NT} - \texttt{NT} - \texttt{NT} - \texttt{NT} - \texttt{T})_{\times 4},$$

The correct evolution for the prediction counter is given shown in Figure 6 (b).

4.2 General case

In the general case (a loop with N iterations repeated M times), the number of mispredictions would be (over-) estimated as $(2M + 1)$: the loop would considered as executing once with two iterations, then $(M - 2)$ times with a single iteration and finally once with $(M(N - 1))$ iterations. As said before, the correct value is $(M + 1)$.

The error comes from how flow information is expressed. The IPET formulation of the problem specifies that the edge that enters the inner loop is executed at most $M \times N$ times. This does not completely reflects the program semantics because the maximum number of iterations for each execution of the loop (*i.e.* N) is not specified. This missing parameter lets the ILP solver finding an infeasible path. Similar observations were mentioned for instruction cache analysis in [14]. However, the problem is more complex for branch prediction because it not possible to establish a direct link between paths in the CFG and transitions in the prediction counter finite-state automaton. In the next section, we show how additional constraints in the ILP model can control the branch predictor behavior for repeated loops.

5 Enforcing valid execution patterns for nested loops

5.1 Extended model

In [14], the worst-case miss rate for an instruction cache is evaluated by considering the possible states of cache lines. The state of a given cache line is the memory block it contains at some point of the program. The way this state changes is expressed by a Cache Conflict Graph (CCG). To some extent, a CCG plays the same role as the branch prediction counter automaton (shown in Figure 3). However, every edge in a CCG can be related to one path in the Control Flow Graph because each node of the CCG is related to a part of the code, *i.e.* to a node in the CFG. On the contrary, nodes in the branch prediction automaton do not stand for code parts and an edge in this automaton might correspond to several control flows.

To illustrate this, let us consider the transition from state `10` to state `01`. This transition can be fired either when the loop is entered (*i.e.* after block `b2` has been executed) or when it iterates (*i.e.* after `b4`). Then it is not possible to directly bound the execution count of this transition ($x^{10}_{\overset{0}{i\rightarrow}}$) to the execution counts of blocks `b2` and `b4` (this was the solution proposed in [14]).

In the case of a branch predictor, if a loop iterates at most N times, the prediction counter cannot fall into the `00` state more than N times it leaves this state. This guarantees that no more than N iterations are considered for each execution of the loop. This can be expressed by this additional constraint:

$$x^{01}_{\overset{0}{i\rightarrow}} + x^{00}_{\overset{0}{i\rightarrow}} + x^{00}_{\Rightarrow i} \leq N \times \left(x^{00}_{\overset{1}{i\rightarrow}} + x^{00}_{\overset{1}{i\Rightarrow}} \right)$$

This constraint applies to any loop with an upper-bounded number of iterations (which means that the effective number of iterations for one execution of the loop might range from 0 to N). This includes triangular loops where the number of executions of the inner loop depends on the value of the iteration counter of the outer loop.

This kind of constraint has to be generated for every block identified as controlling a loop. In the next section, we will give an overview of how the blocks that control loops can be identified from the CFG.

Considering the anomaly in the misprediction counts pointed out in this article, this constraint eliminates from the analysis some infeasible *explicit* paths related to valid *implicit* path. This is likely to tighten the WCET estimation.

5.2 Detecting loop-control blocks

As said before, integrating branch prediction into the IPET model makes it possible to consider various loop constructs. Our model fits different loop patterns (control at the beginning or at the end of the loop, exit of the loop either with a taken or not taken branch) as well as loops with multiple exits (however, to save room, we only describe the constraints for loops with a single exit).

This makes it necessary to identify in the CFG the blocks that contain a loop branch. Our algorithm implements predominance analysis to build sets of blocks that belong to a same loop. Then it searches, in each set, the block that has a successor out of the set: this block is the one that controls the loop and the direction of the branch to exit the loop is determined.

Once the block $\mathtt{b}_{\mathtt{ctrl}}$ that controls a loop has been identified, the number of executions of the loop is $x_{ctrl\overset{1}{\rightarrow}}$ (provided the loop is exited when the branch is taken).

5.3 Experimental results

We have made some experiments to measure the improvement due to refined branch prediction modeling. We have considered four benchmarks from the SNU suite [3]. The Control Flow Graphs of the programs were extracted and analyzed to identify the blocks that control loops using the OTAWA tool [7]. The block execution times were obtained using a cycle-level simulator that models a superscalar out-of-order processor and was developed in our team. Finally, the specification of the ILP problem for WCET calculation (*i.e.* the objective function and the structural and flow constraints) was produced by a perl script. We used `lp_solve` to solve the problem.

To estimate the impact of the infeasible paths on the calculated WCET, we have analyzed the benchmarks with both models: the earlier one, and the extended one proposed in this paper. Results are given in Table 1.

benchmark	old WCET	new WCET
`matmul`	3,246	3,078
`ludcmp`	11,411	11,201
`insertsort`	2,012	1,976
`crc`	211,628	210,098

Table 1. Estimated WCET.

It can be observed that the extended model gives tighter WCETs for all of the benchmarks we have tested. The improvement ranges from 0.72% to 5.17%. In every case, the earlier model over-estimates the number of branch mispredictions and then accounts for superfluous penalties.

6. Conclusion

Modeling advanced processor features using Integer Linear Programming and integrating the model to WCET estimation by IPET has several advantages: most of the loop

patterns can be analyzed, per-branch misprediction penalties can be accounted for, conditional structures are naturally analyzed. However, considering *implicit* paths is not sufficient to analyze schemes that behave according to the execution history. In the case of nested loops, cache miss or branch misprediction counts are overestimated because they are maximized for infeasible explicit paths. To get round this difficulty, we propose an extension to the branch predictor model. Experimental results show that the obtained WCET is tighter.

References

[1] I. Bate and R. Reutemann. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *16th Euromicro Conference on Real-Time systems*, 2004.

[2] I. Bate and R. Reutemann. Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis. In *IEEE Conference on Real-Time Computing Systems and Applications*, 2005.

[3] SNU benchmark suite. http://archi.snu.ac.kr/realtime/benchmark/.

[4] C. Burguière and C. Rochange. A Contribution to Branch Prediction Modeling in WCET Analysis. In *Conference on Design, Automation and Test in Europe (DATE)*, 2005.

[5] C. Burguière and C. Rochange. Modélisation d'un prédicteur de branchement bimodal dans le calcul du WCET par la méthode IPET. In *13th International Conference on Real-Time Systems*, 2005.

[6] C. Burguière, C. Rochange, and P. Sainrat. A Case for Static Branch Prediction in Real-Time Systems. In *IEEE Conference on Real-Time Computing Systems and Applications*, 2005.

[7] H. Cassé and P. Sainrat. OTAWA, a Framework for Experimenting WCET Computations. In *3rd European Congress on Embedded Real-Time Software*, 2006.

[8] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for Processors with Branch Prediction. *Real-Time Systems*, 18(2-3), 2000.

[9] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Euromicro Conference on Real-Time Systems*, 2003.

[10] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards Industry-Strength Worst-Case Execution Time Analysis. Technical Report 99/02, ASTEC, 1999.

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

[12] X. Li, T. Mitra, and A. Roychoudhury. Modeling Control Speculation for Timing Analysis. *Real-Time Systems*, 29(1), 2005.

[13] Y.-T. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. *ACM SIGPLAN Notices*, 30(11), 1995.

[14] Y.-T. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *IEEE Real-Time Systems Symposium*, 1997.

[15] S.-S. Lim, S. Min, M. Lee, C. Park, H. Shin, and C. S. Kim. An Accurate Instruction Cache Analysis Technique for Real-Time Systems. In *Workshop on Architectures for Real-Time Applications*, 1994.

[16] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2), 1999.

[17] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2), 1989.

[18] J. Smith. A Study of Branch Prediction Strategies. In *8th International Symposium on Computer Architecture*, 1982.

A Definition and Classification of Timing Anomalies

Jan Reineke[1], Björn Wachter[1], Stephan Thesing[1], Reinhard Wilhelm[1],
Ilia Polian[2], Jochen Eisinger[2], and Bernd Becker[2]

[1] Saarland University
Im Stadtwald - Gebäude E1 3
66041 Saarbrücken, Germany
{reineke|bwachter|thesing|wilhelm}@cs.uni-sb.de

[2] Albert-Ludwigs-University
Georges-Köhler-Allee 51
79110 Freiburg, Germany
{polian|eisinger|becker}@informatik.uni-freiburg.de

Abstract. Timing anomalies are characterized by counterintuitive timing behaviour. A locally faster execution leads to an increase of the execution time of the whole program. The presence of such behaviour makes WCET analysis more difficult: It is not safe to assume local worst-case behaviour wherever the analysis encounters uncertainty.
Existing definitions of timing anomalies are either given as an intuitive description or do not cover all kinds of known timing anomalies. After giving an overview of related work, we give a concise formal definition of timing anomalies. We then begin to identify different classes of anomalies. One of these classes, coined Scheduling Timing Anomalies, coincides with previous restricted definitions.

Keywords: Timing analysis, Worst-case execution time (WCET), Timing anomalies, Abstraction

1 Introduction

The notion of timing anomalies was introduced by Lundqvist and Stenström in [LS99]. Intuitively, a timing anomaly is a situation where the local worst-case does not entail the global worst-case. For instance, a cache miss – the local worst-case – may result in a shorter execution time, than a cache hit, because of scheduling effects. See Figure 1 for an example. Shortening task A leads to a longer overall schedule, because task B can now block the "more" important task C. Analogously, there are cases where a shortening of a task leads to an even greater decrease in the overall schedule. Such effects are not relevant for timing analysis. We will not consider them in this paper.

Another example occurs with branch prediction. A mispredicted branch results in unnecessary instruction fetching that destroys the cache state. If the first instruction being fetched is a cache miss, the correct branch condition will be computed before more harm can be done by further fetches. Figure 2 illustrates this.

2 Existing Work on Timing Anomalies

The first paper remotely related to timing anomalies was written as early as 1969 by Graham [Gra69]. They show that a greedy scheduler can produce a longer schedule, if provided with

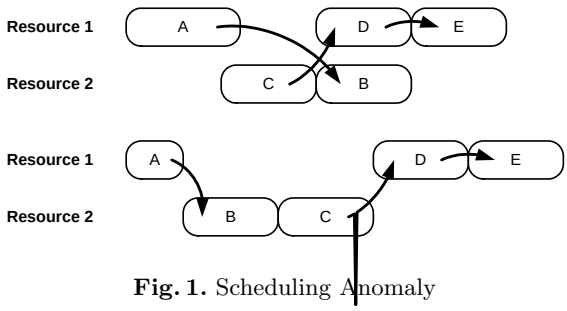


Fig. 1. Scheduling Anomaly

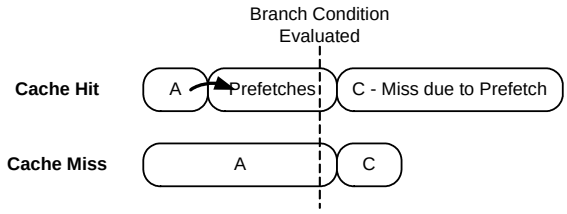


Fig. 2. Speculation Anomaly

shorter tasks, less dependencies, more processors, etc. They also give bounds on these effects, which are known as scheduling anomalies today. In their model all resources (processors) are identical though, which renders the given bounds useless for our purposes.

Lundqvist & Stenström first introduced timing anomalies in roughly the sense relevant for timing analysis. In their 1999 paper [LS99] they give an example of a cache miss resulting in a shorter execution time than a cache hit. A timing anomaly is characterized as a situation where a positive (negative) change of the latency of the first instruction by i cycles results in a global decrease (increase) of the execution time of a sequence of instructions. Situations where the local effect is even accelerated are also considered timing anomalies, i.e. the global increase (decrease) of the execution time is greater than the local change. We do not consider such cases here because they do not pose problems for timing analysis.

In his PhD thesis [Eng02] and in a paper with Jonsson [EJ02], Engblom briefly mentions timing anomalies. He translates the notion of timing anomalies of the Lundqvist/Stenström paper [LS99] to his model by assuming that single pipeline stages take longer, in contrast to whole instructions. Both Lundqvist and Engblom claim that, in processors containing in-order resources only, no timing anomalies can occur. This is not always true unfortunately, as corrected in Lundqvist's thesis [Lun02]. Schneider [Sch02] and Wenzel et al. [Wen03,WKPR05] note that if there exist several resources that have overlapping, but not equal abilities, timing anomalies can also occur.

Thesing [The04] discusses the Motorola ColdFire 5307, which contains a rather simple in-order pipeline that does not even have resources with overlapping abilities. He shows that the processor exhibits timing anomalies caused by its cache. The cache replacement policy of the ColdFire, Pseudo-Round Robin, causes these problems: In contrast to common replacement strategies, such as LRU or Pseudo-LRU, the effect of a cache miss on the cache state is sometimes different from that of a cache hit. While the cache miss obviously consumes a

longer processing time, it may result in a cache state that better suits the following code.

Wenzel, Kirner, Puschner, and Riedel [Wen03,WKPR05] give a necessary condition for timing anomalies, the *Resource Allocation Criterion*, short RAC. The RAC states that it has to be possible to create different schedules for at least one of the functional units of the processor for timing anomalies to be possible. Unfortunately, the criterion is based on a rather restricted definition of timing anomalies. he underlying assumption is that the latencies of subsequent instructions depend solely on the chosen schedule, i.e. which functional units are used. They are assumed to be independent of the initial latency difference. As we have observed in our introductory examples, this is overly optimistic. Both speculation and certain cache replacement strategies, like Pseudo-Round Robin violate this assumption.

3 Formal Definition

While the introductory examples give a rough intuition of what we consider a timing anomaly, they do not offer a concise formal definition. Let us identify important concepts that should flow into a formalization. It should not only cover presently known anomalies but be general enough to be valid also for future hardware features. This desired generality obviously requires a rather abstract approach.

Hardware Model A definition of timing anomalies has to take into account the hardware model. It has a great impact on the number and kind of such anomalies. For instance, out-of-order processors probably show more anomalies than simpler in-order machines.

Timing anomalies require choice, i.e. non-determinism in the analyzed model. In previous work on timing anomalies the different cases to compare, like cache hit or cache miss, came out of the blue.

Abstraction The reason for non-determinism in timing models is abstraction. Timing analysis usually only becomes feasible through abstraction. It enables us to deal with unknown input data and huge state-spaces. In return, abstraction has to give up some precision. Unknown information due to abstraction introduces non-determinism, where the underlying hardware model was fully deterministic. Depending on the precision of the abstraction different timing anomalies are conceivable.

Locality In most examples we have some intuition to what is the local worst-case. To identify a local worst-case formally we need a notion of locality. In literature, locality was usually not explicitly treated, but often implicitly fixed to the instruction level [LS99,Lun02]. Engblom [Eng02,EJ02] considered micro-operations (pipeline stages) to be the right granularity. We believe that micro-operations (like instruction fetch, execute, etc.) are indeed the right locality level. This is where timing differences first become visible.

Based on these observations we will now formally define timing anomalies. Our definition requires some notational prerequisites:

Definition 1 (Transition System). *A* transition system T *is a pair* $T = (S, R)$*, where* S *is a finite set of states and* $R \subseteq S \times S$ *is a transition relation. A* path π *in a transition system* $T = (S, R)$ *is a finite sequence of states, s.t.* $(\pi_i, \pi_{i+1}) \in R$ *for all* $i \in 0 \ldots |\pi| - 1$*. The* set of all paths *of a transition system* T *is denoted as* $\Pi(T)$*.*

A transition system can model the cycle-level behaviour of a computer architecture, i.e. a transition models the execution of one cycle. In contrast to other low-level hardware models,

such as Mealy- or Moore-automata, inputs and outputs are not explicitly modeled. This is not necessary in our context of timing analysis. Data and the program that is executed are modelled as part of the state.

As noted above, we consider micro-operations to be the right level to make local decisions, i.e. identify the local worst-case. The following definition of *locality constraints* enables us to do so.

Definition 2 (Locality Constraint). *A* locality constraint *l for a transition system $T = (S, R)$ is a convex predicate on S, i.e. l only holds on consecutive states in any path π through T. We assume that* locality constraints *model the sequence of states that is executing a micro-operation. We denote the* restriction *of π to l by $\pi_{|l}$, i.e. the restriction of the path π to the subpath of π in which l holds. Note, that this is still a (possibly empty) path. We denote the restriction of π to a set $\{l_1, \ldots, l_n\}$ of locality constraints by $\pi_{|l_1 \ldots l_n}$, i.e. the restriction of the path π to a sequence of states of T in which at least one of the predicates $l_1, \ldots, l_n$ holds. $\pi_{|l_1 \ldots l_n}$ is not necessarily a path.*

Definition 3 (Local Worst-Case Path). *Given a set of locality constraints $\mathcal{L}$ and a set of paths Π, a path $\pi \in \Pi$ is a* local worst-case path*, if and only if for every locality constraint $l \in \mathcal{L}$ and every path $\pi' \in \Pi$ it holds that if $\pi = \pi_{pre} \circ \pi_{|l} \circ \pi_{post}, |\pi_{|l}| > 0$ and $\pi' = \pi_{pre} \circ \pi'_{|l} \circ \pi'_{post}$ then $|\pi_{|l}| \geq |\pi'_{|l}|$.*
A path is called a non-local worst-case path *if it is not a* local worst-case path*.*

Definition 4 (Program). *A* program *(or a* control flow graph*) P is a directed graph $P = (V, E), E \subseteq V \times V$, in which the nodes V represent instructions, and an edge $(u, v) \in E$ represents flow of control from u to v.*
A sequence *σ through a program $P = (V, E)$ is a finite sequence of instructions, s.t. $(\sigma_i, \sigma_{i+1}) \in E$ for all $i \in 0 \ldots |\sigma| - 1$.*

We do not want to compare arbitrary paths through the transition system, but only those that correspond to the same path through the program. We can map paths in the transition system to paths through the program via a *labelling function*.

Definition 5 (Labelling Function). *Given a transition system $T = (S, R)$, a set of locality constraints $\mathcal{L}$, and a program $P = (V, E)$. A* Labelling Function *$\rho : V^* \to \mathcal{P}(\mathcal{L})$ assigns each finite sequence of instructions through the program P a set of locality constraints that corresponds to the execution of the respective micro-operations. A path π through T then corresponds to the sequence σ through P iff $\pi_{|l}$ is not empty for all $l \in \rho(\sigma)$ and $\pi_{|\rho(\sigma)}$ is equal to π.*
Given a set Π of paths through T, the subset of Π which corresponds to a given sequence σ w.r.t. a labelling function ρ is denoted as $\Pi|^{\rho}_{\sigma}$.

Definition 6 (Hardware Model). *A (possibly abstracted) hardware model C maps a program P to a transition system T, a set of locality constraints $\mathcal{L}$ on T, and a labelling function ρ that relates the states of the transition system with the instructions of the given program P.*

Note that a concrete hardware model is deterministic. Non-determinism – which is necessary for timing anomalies – is only introduced by abstraction. One can formally define the relation between concrete and abstract hardware models. For reasons of brevity we omit to provide such a definition.
Now, we are ready to define timing anomalies.

Definition 7 (Timing Anomaly). *A hardware model C exhibits timing anomalies, if there exists a program P with $C(P) = (T, \mathcal{L}, \rho)$, a finite sequence σ through P, and a non-local worst-case path $\pi \in \Pi(T)|^{\rho}_{\sigma}$, s.t. $|\pi| > |\pi'|$ for all local worst-case paths $\pi \in \Pi(T)|^{\rho}_{\sigma}$.*

Figure 3 illustrates the situation. At some analysis state, after executing π_{pre}, future execution is non-deterministic. To find the globally longest path we need to follow the non-local worst-case path $\pi_{|l}$ (it is not the longest path on locality constraint l).

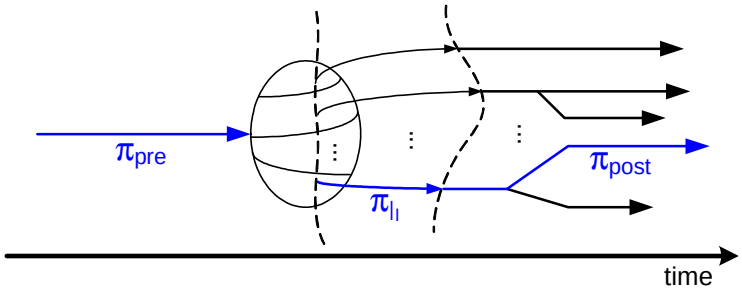


Fig. 3. Timing Anomaly Example

4 Classification

The above definition introduces timing anomalies in a rather abstract way. In the future, when confronted with a possible anomaly it will allow us to safely argue whether or not it constitutes a timing anomaly. This section aims at starting to clarify "what timing anomalies really are", by identifying different subclasses.

The idea is to readopt the view of timing anomalies from a scheduling perspective. In this setting, a set of tasks with dependencies and resource constraints describes the problem posed to the hardware. Tasks could be executions of micro-operations. Dependencies ensure that the micro-operations of a specific instruction can only be executed sequentially. Other dependencies model data dependencies in a program. Resources are stages in the pipeline like Instruction Fetch, Execute, or the different functional units of the processor. Now, we can distinguish at least three classes of timing anomalies (and possibly many more):

Scheduling Timing Anomalies We compare two task sets that differ only in the length of the "pivot" task. An example could be a cache hit vs. a cache miss. Figure 1 gives an example. The task sets differ only in the length of task A. Most timing anomalies dealt with in literature fall into this category. This kind of anomaly is well-known in the scheduling world, and has been extensively studied on various scheduling routines. One observation that can be made is that greedy schedulers, mimicked by timing analysis (and online schedulers, like modern processors, usually are greedy) are unable to prevent such anomalies in general.

Speculation Timing Anomalies Here, the difference is not confined to the length of the "pivot" task. The entire task set changes depending on this task. As an example see Figure 2. In both cases the processor is speculatively prefetching instructions. The local worst-case, a cache miss while fetching the first instruction, takes so much time that the branch condition can be evaluated, before more harm can be done to the cache by

further prefetches. Interestingly, the task set is influenced by previous decisions of the scheduler. Apparently, these interactions put these anomalies outside of the scope of scheduling theory. Note that the anomaly can occur even if the abstraction knows that the branch was mispredicted.

Cache Timing Anomalies These are anomalies induced by strange cache behaviour, as in the Pseudo-Round Robin cache replacement strategy employed in the ColdFire 5307. There, the non-local worst-case cache hit results in a different future cache state than the local worst-case cache miss. The difference in the cache state can then cause the cache hit branch to be stalled later on.

Interestingly, the latter two classes of anomalies can also happen on in-order architectures, as the ColdFire.

5 Conclusion

Timing anomalies result from complex interactions in modern processors and non-determinism introduced by abstraction. Their definition is a difficult task. We have given an overview of existing work on timing anomalies, and identified imprecisions and weaknesses. Notably, the restriction to what we call Scheduling Anomalies and the lack of formalization of locality. Based on these observations, we have given a concise formal definition, that is – unlike previous definitions – general enough to cover all known kinds of anomalies. Furthermore, we have begun to identify different classes of timing anomalies.

References

[EJ02] Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static wcet analysis. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 334–348, London, UK, 2002. Springer-Verlag.

[Eng02] J. Engblom. Processor pipelines and static worst-case execution time analysis, 2002.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[LS99] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Sweden, June 2002.

[Sch02] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, Germany, December 2002.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, Germany, July 2004.

[Wen03] Ingomar Wenzel. Principles of timing anomalies in superscalar processors. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[WKPR05] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference on Quality Software*, Sep. 2005.

PLRU Cache Domino Effects

Christoph Berg
Saarland University, Compiler Design Lab, Saarbrücken, Germany
cb@cs.uni-sb.de

June 9, 2006

Abstract

Domino effects have been shown to hinder a tight prediction of worst case execution times (WCET) on real-time hardware. First investigated by Lundqvist and Stenström, domino effects caused by pipeline stalls were shown to exist in the PowerPC by Schneider. This paper extends the list of causes of domino effects by showing that the *pseudo LRU* (PLRU) cache replacement policy can cause unbounded effects on the WCET. PLRU is used in the PowerPC PPC755, which is widely used in embedded systems, and some x86 models.

1 Introduction

Embedded systems play a key role in any modern product. When employed in safety-critical environments like airbag controllers in cars or fly-by-wire systems in air crafts, the timing must meet conditions imposed by the environment. The execution time of the tasks running on the embedded processor must always be lower than a given deadline. Timing analysis is used to derive an upper bound on the execution time, called *worst case execution time (WCET)*. Computing the WCET of a program requires upper bounds for the number of iterations of all loops in the program. On every path through the program, the worst case execution times for all instructions (or alternatively, basic blocks) has to be computed and added up. The longest of these paths is then called the critical path, and its maximum runtime is the WCET.

1.1 Timing Anomalies

In a simple world, timing analysis could just assume the local worst case for all instructions in a pipeline. When the cache state is not be precisely known, a memory access not classified by the abstract cache state as a cache hit would be considered a cache miss, variable-latency instructions would just take the longest time, etc. We could then add up all individual times and get a safe WCET bound.

Unfortunately, this is not safe. Out-of-order pipelines might reorder instructions such that an longer initial delay (e.g., a cache miss) could cause an overall *faster* completion of the whole sequence. Similarly, a speedup of an instruction can lead to a longer runtime for the whole sequence. This effect is called a *timing anomaly*, and was first described by Lundqvist and Stenström [3, 2].

While we can easily get a rough intuition about what a timing anomaly is, a general, hardware-independent definition is difficult, see [4] for a recent result. We will not detail the definition in this paper.

1.2 Domino Effects

A special case of timing anomalies is the *domino effect*, where – after a (possibly empty) prologue – a sequence of instructions is executed in a loop and depending on the initial state of a component (usually the pipeline, but also the cache, as we will see later), the loop body runtime will take different values without convergence. The presence of domino effects means that we cannot unroll a bounded (or even any) number of iterations of a loop and assume in the analysis that the remaining iterations behave the same. Schneider was able to demonstrate actual domino effects caused by the PowerPC PPC755 pipeline [5].

2 Domino Effects in Caches

We will look at cache behavior when a sequence of memory accesses is repeated indefinitely. We will only consider single cache sets.[1]

2.1 FIFO

FIFO caches require very little update logic, a round-robin counter points the next way to be replaced. The downside is that this causes domino effects.

Figure 1 shows a 2-way FIFO cache with the access sequence a-b-c. Starting with an empty cache leads to a repeated cache content b-c at the end of each iteration, where each cycle has 3 cache misses (marked by ‘x’ in the figure). Starting with c-a in the cache leads to a cycle over 2 iterations with a total of 3 cache misses, alternatingly 1 and 2 per iteration.[2]

[1] There are cache architectures where the sets are not independent, but that only makes timing effects more unpredictable.

[2] This is the same example as in [2].

	. .		c a	
a:	a .	x	c a	
b:	b a	x	b c	x
c:	c b	x	b c	
a:	a c	x	a b	x
b:	b a	x	a b	
c:	c b	x	c a	x
a:	a c	x	c a	
b:	b a	x	b c	x
c:	c b	x	b c	
a:	a c	x	a b	x
b:	b a	x	a b	
c:	c b	x	c a	x

Figure 1: 2-way FIFO cache, empty cache is worst case

2.2 LRU

The LRU update logic is complex, and LRU caches with more than 4 ways are rare in practice.

LRU caches do not exhibit domino effects. When iterating a over sequence of instructions, it is easy to see that the cache contents at the end of the first iteration are the same as at the end of every other iteration. This "memory-less" property of LRU makes the cache analysis both easy and precise [1].

2.3 PLRU

Pseudo LRU replacement is a variant of LRU where the "ages" of the lines in the cache are not linearly ordered, but arranged in a tree (see Fig. 2). The advantage is that this needs fewer state bits and hence needs a less complex update logic. PLRU has an average case performance comparable to LRU, but the worst case performance, which matters for the WCET prediction, is worse.

Heckmann has shown that only 3 to 4 ways of an 8-way PLRU cache can be tracked in cache analysis [1]. This article adds domino effects to the list of problems with analyzing PLRU caches.

PLRU definition. PLRU maintains a tree of cache ways. Every inner tree node has a bit pointing to the subtree that contains the leaf to be replaced next. Figure 2 shows a 4-way example. In the left picture, the three state bits b_0, b_1, b_2 point to the second way to be replaced next. In practice, 8-way PLRU is used. The replacement logic is the same, with 7 state bits, and an additional level in the tree. For simplicity, we consider 4-way in this article. Note that 2-way PLRU is equivalent to 2-way LRU.

On a cache update, and similarly on a cache hit, the state bits on the path leading to the accessed way will be flipped, i.e. making them pointing away from that way. The right picture shows the cache state after the replacement of 'b' by 'e'. The path to the second way consists of bits b_0 and b_1, so these have been flipped.

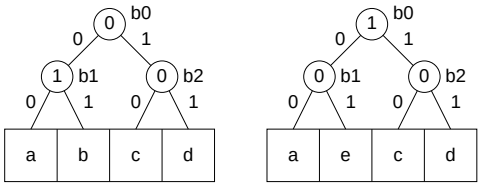


Figure 2: 4-way PLRU

	a b c d	0 0 0	
a:	a b c d	1 1 0	
e:	a b e d	0 1 1	x
a:	a b e d	1 1 1	
f:	a b e f	0 1 0	x
a:	a b e f	1 1 0	
g:	a b g f	0 1 1	x
a:	a b g f	1 1 1	
e:	a b g e	0 1 0	x
a:	a b g e	1 1 0	
f:	a b f e	0 1 1	x
a:	a b f e	1 1 1	
g:	a b f g	0 1 0	x
a:	a b f g	1 1 0	
e:	a b e g	0 1 1	x
a:	a b e g	1 1 1	
f:	a b e f	0 1 0	x
a:	a b e f	1 1 0	
g:	a b g f	0 1 1	x

Figure 3: 4-way PLRU cache, b is never evicted

PLRU examples. Figure 3 shows that PLRU does not use the cache optimally. The access sequence a-e-a-f-a-g contains only 4 distinct elements, yet 4-PLRU misses for every second access when started with a-b-c-d in the cache.[3]

Figure 4 shows that a PLRU cache can take several iterations to stabilize. The fourth iteration is the first that exhibits 3 misses as all the following iterations do.

Figure 5 shows an example with the access sequence i-f-e-g-i-e-b. The left sequence starts with a-b-c-d in the cache, the right one with an empty cache. For the first two iterations, both caches miss on the same accesses (the cache content is even the same at the end of the first iteration, though in a different order). Starting from the third iteration, the left cache misses 3 times per iteration, the right cache only 2 times.

2.4 Avoiding Domino Effects

There is no generally safe initial cache state in the context of domino effects as any state can cause non-converging loop runtimes. With a given program and initial state, we can prove the presence or absence of effects, but in many embedded system context, modifying the program is infeasible, and the initial state can also be unknown or undefined. Modifying the

[3] Example similar to Fig. 3 in [1].

	b e c a	0 0 0	
c:	b e c a	0 0 1	
a:	b e c a	0 0 0	
b:	b e c a	1 1 0	
e:	b e c a	1 0 0	
b:	b e c a	1 1 0	
c:	b e c a	0 1 1	
d:	b d c a	1 0 1	**x**
c:	b d c a	0 0 1	
a:	b d c a	0 0 0	
b:	b d c a	1 1 0	
e:	b d e a	0 1 1	**x**
b:	b d e a	1 1 1	
c:	b d e c	0 1 0	**x**
d:	b d e c	1 0 0	
c:	b d e c	0 0 0	
a:	a d e c	1 1 0	**x**
b:	a d b c	0 1 1	**x**
e:	a e b c	1 0 1	x
b:	a e b c	0 0 1	
c:	a e b c	0 0 0	
d:	d e b c	1 1 0	**x**
c:	d e b c	0 1 0	
a:	d a b c	1 0 0	x
b:	d a b c	0 0 1	
e:	e a b c	1 1 1	x
b:	e a b c	0 1 1	
c:	e a b c	0 1 0	
d:	e d b c	1 0 0	x
c:	e d b c	0 0 0	
a:	a d b c	1 1 0	x
b:	a d b c	0 1 1	
e:	a e b c	1 0 1	x
b:	a e b c	0 0 1	
c:	a e b c	0 0 0	
d:	d e b c	1 1 0	x

Figure 4: 4-way PLRU cache, convergence in the forth cycle

compiler not to produce code exhibiting domino effects seems impossible, given the simplicity of our FIFO examples.

At first glance, it might seem that for PLRU, we can improve our knowledge of the cache state by not just taking into account the ordering of the blocks in the cache but also include the tree state bits. However, we can reorder the ways such that the tree bits always point left, and modify the cache update accordingly. As with FIFO caches, we will have domino effects in this simplified cache.

3 Summary

Timing anomalies and domino effects cause the complexity of timing analysis to grow. Several causes are known, this article adds PLRU caches to the list.

To achieve better predictability, some embedded system designers lock all but two PLRU ways, leaving a 2-way LRU subset. We are currently working on cache models that will hopefully be able to extract more precise information from a non-locked cache despite the presence of timing anomalies.

	a b c d	0 0 0		. . . .	0 0 0	
i:	i b c d	1 1 0	x	i . . .	1 1 0	x
f:	i b f d	0 1 1	x	i f . .	1 0 0	x
e:	i e f d	1 0 1	x	i f e .	0 0 1	x
g:	i e f g	0 0 0	x	i f e g	0 0 0	x
i:	i e f g	1 1 0		i f e g	1 1 0	
e:	i e f g	1 0 0		i f e g	0 1 1	
b:	i e b g	0 0 1	x	i b e g	1 0 1	x
i:	i e b g	1 1 1		i b e g	1 1 1	
f:	i e b f	0 1 0	x	i b e f	0 1 0	x
e:	i e b f	1 0 0		i b e f	0 1 1	
g:	i e g f	0 0 1	x	i g e f	1 0 1	x
i:	i e g f	1 1 1		i g e f	1 1 1	
e:	i e g f	1 0 1		i g e f	0 1 1	
b:	i e g b	0 0 0	x	i b e f	1 0 1	x
i:	i e g b	1 1 0		i b e f	1 1 1	
f:	i e f b	0 1 1	**x**	i b e f	0 1 0	
e:	i e f b	1 0 1		i b e f	0 1 1	
g:	i e f g	0 0 0	x	i g e f	1 0 1	x
i:	i e f g	1 1 0		i g e f	1 1 1	
e:	i e f g	1 0 0		i g e f	0 1 1	
b:	i e b g	0 0 1	x	i b e f	1 0 1	x
i:	i e b g	1 1 1		i b e f	1 1 1	
f:	i e b f	0 1 0	**x**	i b e f	0 1 0	
e:	i e b f	1 0 0		i b e f	0 1 1	
g:	i e g f	0 0 1	x	i g e f	1 0 1	x
i:	i e g f	1 1 1		i g e f	1 1 1	
e:	i e g f	1 0 1		i g e f	0 1 1	
b:	i e g b	0 0 0	x	i b e f	1 0 1	x

Figure 5: 4-way PLRU cache, domino effect starting in the third iteration

References

[1] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proc.*, 91(7), July 2003.

[2] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, School of Computer Science and Engineering, Chalmers University of Technology, 2002.

[3] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. Number 20 in IEEE Real-Time Systems Sym. (RTSS'99), Dec. 1999.

[4] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, July 2006.

[5] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universität des Saarlandes, Dec. 2002.

Design of a WCET-Aware C Compiler *

Heiko Falk
Computer Science 12
University of Dortmund
D-44221 Dortmund
Heiko.Falk@udo.edu

Paul Lokuciejewski
Computer Science 12
University of Dortmund
D-44221 Dortmund
Paul.Lokuciejewski@udo.edu

Henrik Theiling
AbsInt Angewandte Informatik
Science Park 1
D-66123 Saarbrücken
theiling@absint.com

Abstract

This paper presents techniques to tightly integrate worst-case execution time (WCET) information into a compiler framework. Currently, a tight integration of WCET information into the compilation process is strongly desired, but only some ad-hoc approaches have been reported currently. Previous publications mainly used self-written WCET estimators with very limited functionality and preciseness during compilation. A very tight integration of a high quality industry-relevant WCET analyzer into a compiler was not yet achieved up to now. This work is the first to present techniques capable of achieving such a tight coupling between a compiler and the WCET analyzer aiT. This is done by automatically translating the assembly-like contents of the compiler's low-level intermediate representation (LLIR) to aiT's exchange format CRL2. Additionally, the results produced by the WCET analyzer are automatically collected and re-imported into the compiler infrastructure. The work described in this paper is smoothly integrated into a C compiler environment for the Infineon TriCore processor. It opens up new possibilities for the design of WCET-aware optimizations in the future.

The concepts for extending the compiler infrastructure are kept very general so that they are not limited to WCET information. Rather, it is possible to use our structures also for multi-objective optimization of e. g. best-case execution time (BCET) or energy dissipation.

1. Introduction

In contrast to general-purpose systems, embedded systems often have to meet some real-time constraints making them real-time systems. The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced [5]. Besides the necessity of safeness of real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing of a real-time application, the designer tends to use oversized hardware in order to guarantee the safeness of the real-time system. The knowledge of the worst-case execution time *(WCET)* gives the designer the opportunity to use or to design a hardware platform which is tailored towards the software resource requirements like memory or clock rate. Thus, the production costs can be reduced significantly, while still guaranteeing the safeness of the real-time system.

Today, software development for embedded systems is done using high-level languages like C, requiring the existence of a suitable compiler. Modern compilers are equipped with a vast variety of optimizations. However, these optimizations aim at minimizing e. g. average-case execution time *(ACET)* [14] or energy dissipation [15]. The influence of compiler optimizations on WCET is unknown in almost all cases. Currently, the binary executable generated by the compiler is manually fed into a WCET analyzer computing the required information. Using this WCET data, it can be determined whether real-time constraints are met. If this is not the case, the program source code has to be tuned, compiled and optimized again in another cycle of the design flow.

As can be seen, it is highly desirable to have a compiler that is aware of WCET properties. In an integrated WCET-aware compilation environment, it will be possible to integrate and to apply optimizations with the objective of WCET minimization. WCET information available within the compiler can be used to determine those parts of the code that lie on the worst-case path. Specialized complex optimizations could be applied in the future only to these code portions in order to minimize WCET aggressively. If the compiler is able to support multiple optimization objectives at the same time (e. g. energy consumption and WCET), automated trade-offs can be applied by the compiler such that the most energy efficient code is generated that still meets real-time constraints.

This paper is the first one to present a tight integration of an existing WCET analyzer into a compiler infrastructure. Using the techniques of this work, it is possible to feed the assembly-like contents of the compiler's low-level intermediate representation *(LLIR)* into the WCET analyzer automatically. The results produced by the WCET analyzer are automatically re-imported into the compiler infrastructure and are available for future optimizations as described above. Currently, this includes the WCET of an entire application, of functions and basic blocks as well as calling contexts, execution counts and feasibility information. The techniques presented in this work are part of a C compiler environment and were validated for the Infineon TriCore [9] processor.

In addition, a very powerful and flexible mechanism is presented enabling to attach not only WCET-related data to the LLIR, but also to store arbitrary information used by optimizations targeting different objectives than WCET. This approach will be useful in order to perform automated trade-offs between different optimization goals.

The remainder of this paper is structured as follows: Section 2 gives a survey of related work. Section 3 presents the entire structure of our WCET-aware C compiler. It is fol-

*This work is partially funded by the European IST FP6 Network of Excellence ARTIST2.

lowed by a discussion on the proposed compiler structure in Section 4. Finally, Section 5 summarizes this paper and gives an outlook on future work.

2. Related Work

[4] presented a very first simple approach to integrate WCET techniques into a commercial compiler. Flow facts required for WCET analysis have to be annotated manually using source-level pragmas. The compiler backend generates code for the Intel 8051 which is an inherently simple and predictable machine without pipeline and caches etc. The entire work was not finished and tested, and results are unavailable. A fully pragma based approach is not promising since manual annotations are tedious and error-prone.

While mapping high-level code to object code, compilers perform various optimizations so that the correlation between high-level flow facts and the optimized object code becomes very vague. In order to keep track of the influence of compiler optimization on high-level flow facts, [7] proposes co-transformation of flow facts. However, the co-transformer has never reached a fully working state, and several standard compiler optimizations can not be modeled at all due to insufficient data structures.

[12] presents techniques for transforming program path information during compiler optimization. The authors are able to keep high-level flow facts consistent while performing GCC's standard optimizations. Their approach was thoroughly tested and led to precise WCET estimates. However, compilation and WCET analysis are done in a decoupled way. The assembly file generated by the compiler is passed to the WCET analyzer together with the transformed flow facts. Additionally, the proposed compiler is only able to process a subset of the C programming language, and the modeled target processor lacks pipelines and caches.

In [17, 18], the integration of a proprietarily developed WCET analyzer into a compiler is presented. The compiler operates on a low-level intermediate representation *(IR)*. Control flow information is passed to the timing analyzer which computes the WCET of loops and functions and passes this data back to the compiler. However, this approach has several limitations. First, the WCET analyzer works with very coarse granularity since it only computes WCETs of loops and functions. WCETs for basic blocks or single instructions are unavailable. Second, WCET-relevant data which is not the WCET itself is unavailable, too. This includes e. g. execution contexts, execution frequencies of basic blocks, value ranges of registers, predicted cache behavior etc. Third, the lack of a high-level IR within the compiler requires to costly re-generate valuable high-level flow facts that are only available at the source code level. Finally, the considered processor is quite simple as it has a simple pipeline and no caches.

A compiler guided trade-off between WCET and code size is presented in [13]. The authors observe that applications using the 16-bit THUMB instruction set of an ARM7 processor typically are smaller but run slower than when using the 32-bit ARM instruction set. [13] compiles a program using both instruction sets at the same time with the goal to reduce WCET at the expense of code size and vice versa. To obtain WCET information, a simple timing analyzer was implemented from scratch assuming the absence of caches and a simple pipeline structure. [13] presents a

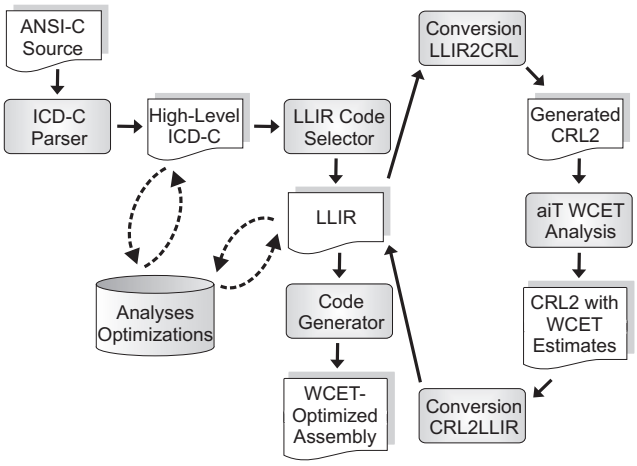


Figure 1. WCET-aware C Compiler (WCC)

WCET-guided compiler optimization, but it does not homogeneously integrate WCET analysis into the compiler itself. Compilation and WCET analysis are completely decoupled, WCET data is not fed back to the compilation stages.

An optimization allocating both functions and data elements of an application to a scratchpad memory is presented in [16]. The authors report a significant WCET reduction, since scratchpads are much faster than other types of memory, and their use is fully guided by the compiler so that very precise flow facts can be passed to the WCET analyzer. However, this approach also does not establish a tight integration of WCET data into the compiler. The memory allocation optimization is not based on WCET data. Instead, the optimization minimizes energy dissipation when using scratchpad memories. WCET timing is measured after compilation by analyzing the resulting binary executable.

3. Design of the WCET-Aware C Compiler

Based on the results of the previous publications presented in Section 2, we propose the structure depicted in Figure 1 for WCET-aware compilation. As can be seen, the compiler we call *WCC* relies on two IRs, namely ICD-C [10] being close to ANSI-C, and LLIR [11] modeling code at the assembly level. The code selector is responsible for the translation of ICD-C to LLIR. Analyses and optimizations take place both at the high and the low level. Currently, all optimizations done by the compiler are not yet WCET-aware. The focus of this paper clearly lies on the integration of WCET analysis into the compiler, and not on WCET optimizations which are part of our future work.

Today, there are only few vendors providing commercial WCET analyzers. One of the leading analyzers is the aiT [1] WCET analyzer developed by AbsInt. aiT is available for various processors, including ARM7, Power-PC, TI TMS320C33 and the Infineon TriCore v1.3. In this paper, the Infineon TriCore processor is considered as target architecture for compilation and WCET analysis.

WCET analysis can only take place at the assembly / binary level since processor-specific information and machine code is unavailable at higher levels of abstractions. Thus, the WCET analyzer aiT is coupled to our compiler at the LLIR level. Our compiler is able to translate its LLIR representation to the CRL2 format [2] which is the IR of aiT. This way, both aiT and WCC use the same format to ex-

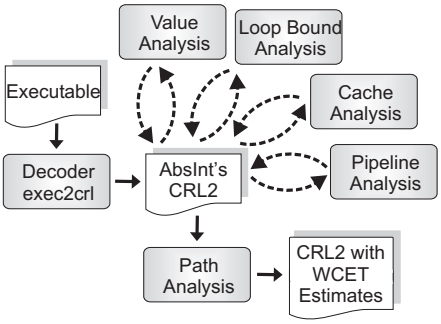


Figure 2. Workflow of aiT

change information. Hence, aiT can be provided with CRL2 accurately modeling the application under analysis, and all results computed by aiT are also available in the CRL2 format after WCET analysis is done. Using a conversion from CRL2 to LLIR, we are able to import all data computed by aiT into our compiler.

3.1. The WCET Analyzer aiT

In contrast to numerous other WCET analyzers, aiT performs a highly accurate analysis of the processor pipeline and available caches. When being used outside our WCC compiler framework, aiT needs to be provided with a binary executable of the program to be analyzed as well as with a specification file. To estimate the WCET of the binary executable, several analysis steps are taken (cf. Figure 2). The decoder exec2crl reads the executable and reconstructs its control flow graph. This control flow graph is translated into aiT's intermediate format CRL2 [2]. CRL2 is used as exchange format storing the application under analysis as well as analysis results generated by the individual substeps of aiT. In the WCC setup, aiT's decoder exec2crl is skipped.

Value analysis determines potential values in the processor registers for any possible program point. These values are frequently used within aiT. Cache analysis requires predicted values to identify possible addresses of memory accesses. In addition, the predicted values are used to determine infeasible paths resulting from conditions being true or false at any point of the analyzed program. Finally, tight value ranges are required to determine loop bounds.

Since a program spends most of its execution time in loops, the iteration counts play an important role for WCET estimation. aiT relies on precise bounds to be able to perform a WCET analysis at all. The detection of loop bounds during *loop bound analysis* succeeds only for simple loops and demands their external annotation outside aiT.

The *cache analysis* of aiT statically analyzes the cache behavior of a program using a formal cache model. Accesses to main memory are examined by an algorithm distinguishing between sure cache hits and unclassified accesses. A proper cache analysis relies on the value ranges of processor registers obtained from the value analysis.

Pipeline analysis models the processor's pipeline behavior and is based on the current state of the pipeline, the resources in use, the contents of prefetch queues and the results obtained during cache analysis. It aims at finding a WCET estimate for each basic block of a program. Each basic block is analyzed by taking possible pipeline states from preceding basic blocks into account. After processing each instruction of the currently analyzed block, the longest time this block takes to execute is computed.

Using the data provided by the previous analysis steps, *path analysis* computes a program's global WCET. A path within the control flow graph is a sequence of successive basic blocks starting at the entry point of a program and ending at its end point. For each block on a path, its maximum execution time T can be expressed based on the previous analyses. Using the loop bound information, a block's maximum number of executions C can be determined. The WCET of a path can be expressed as the sum of the products $T * C$ over all edges of a path. A program's global WCET is determined by finding the maximum path WCET for all feasible paths. This maximization problem is expressed and solved using integer linear programming.

All analyses of aiT take *execution contexts* into account. A context indicates a particular way of calling a program's function R. If R is called several times, contexts are used to distinguish between the different program states representing the different flows of control through the program to function R. Contexts improve the precision of analysis if analysis results are computed individually for each program state represented by a context.

Since aiT is directly invoked with a CRL2 IR generated by our compiler, the decoder exec2crl shown in Figure 2 is skipped in our WCC setup. In the following subsections, the conversions between LLIR and CRL2 and vice versa are described as well as a generic mechanism for storing arbitrary data beyond WCET information within LLIR.

3.2. Conversion LLIR ↔ CRL2

Since both LLIR and CRL2 are low-level IRs representing a program as control flow graph *(CFG)*, it is easy to translate these IRs into each other. At the top level, the control flow graph of both IRs consists of functions. Each function contains a list of basic blocks connected via edges. Basic blocks in turn consist of a sequence of instructions. In both IRs, an instruction can consist of several operations in order to express the implicit parallelism present in e. g. a VLIW machine. In addition to basic blocks, information on execution contexts is attached to functions within CRL2. In LLIR, this kind of information does not exist.

Due to the analogy of both IRs, the translation steps basically need to traverse the CFG of one IR recursively and need to generate equivalent CFG components of the other IR. However, there are two issues making the translation from LLIR to CRL2 and back much more complex than this simple recursive traversal.

Determination of *op_id*s

First, we need to solve the problem that LLIR is an assembly-level IR, whereas CRL2 is a binary-level IR. As a consequence, any information additionally created during assembly and linking is unavailable within LLIR, but available to CRL2. This difference between both IRs mainly comes into play when converting LLIR operations into CRL2 operations. An LLIR operation is represented by its assembly mnemonic and belonging operands, whereas CRL2 requires a unique identifier (*op_id*) representing the binary machine code of the operation together with its operands. Unfortunately, there is no direct translation between an LLIR opcode and a corresponding *op_id*.

For example, the TriCore ISA contains four different ma-

chine operations with the mnemonic AND [9]:

AND Dc, Da, Db AND Dc, Da, *const9*

AND Da, Db AND D15, *const8*

All these AND operations differ by the number and types of parameters, namely data registers denoted by Dx or D15, or constants. As can be seen, the mere use of the mnemonic does not yield an unambiguous CRL2 operation. As a consequence, more key characteristics of an LLIR operation need to be taken into account.

In addition to its mnemonic, the operation's *opcode format* is considered next. Depending on the amount and type of parameters, the binary machine code of an operation has various formats. For example, the first AND operation listed above has the 32-bit wide format RR. The second operation with a constant parameter is of format RC. The last two operations with only two parameters have the 16-bit opcode formats SRR and SC, respectively.

Using the alignment and the type of parameters of an LLIR operation, its opcode format is determined. In almost all cases, these first two steps of mnemonic and opcode format matching are sufficient to obtain an unambiguous *op_id*. However, the addressing mode has to be considered as third criterion in a few cases. The addressing mode specifies the calculation of an effective memory address of a certain operation parameter, using values held in registers and constants. Mainly load and store operations make extensive use of the various addressing modes. The TriCore architecture provides eight different modes such as *Pre-* and *Post-increment* or *Short* and *Long Base plus Offset* addressing.

As example, the LD.W operation loading a word from memory is considered. The TriCore instruction set includes only one operation with the opcode LD.W:

LD.W Da, <*mode*>

Its first parameter designates the register to be loaded, the second one specifies the addressing mode. Depending on this mode, various CRL2 *op_id*s can be used for the LD.W operation. It is easy to see that mnemonic and opcode format are insufficient in this case to obtain a unique *op_id*.

But even after considering addressing modes, there are still some operations which can not be unambiguously identified so far. In particular, halfword arithmetical operations have equivalent mnemonics and opcode formats, but do not rely on addressing modes at all. The only difference between these halfword operations is a specifier like e. g. UL denoting that an upper halfword of a register is combined with a lower halfword of another register. However, these few cases of halfword arithmetic can be handled easily by performing particular checks for these halfword operations.

But even after this fourth stage of halfword operation matching, there still exist LLIR operations which are not unambiguously mapped to CRL2 *op_id*s. The TriCore instruction set still contains few operations which can not be classified unambiguously at all using the information available within LLIR. For instance, there are two versions of the MOV operation:

MOV Dc, Db MOV Da, Db

The parameters of these two operations are arbitrary data registers in both cases. However, the first operation is 32 bits wide, whereas the second operation is a 16-bit operation. Since the LLIR does not distinguish the bit-width of operations, it is impossible to classify these two opcode formats correctly. As a consequence, a decision on the underlying opcode format has to be taken during translation from LLIR to CRL2. In such a situation, our compiler framework decides to assume the 32-bit version of an operation.[1] After this decision is finally taken, all LLIR operations are assigned a unique CRL2 *op_id*. Hence, a complete translation of our compiler's LLIR into aiT's IR is achieved.

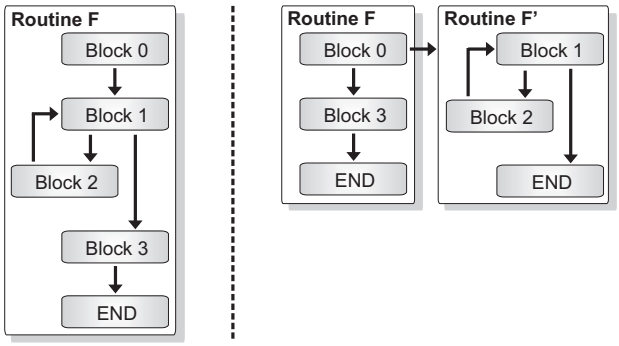


Figure 3. Loop Transformation Stage of aiT

Attention needs to be paid that the decision taken during the previous phase described above is also considered during the subsequent compilation steps. When writing out the contents of the LLIR into an assembly file, a line just like

```
MOV D9, D8
```

will be dumped. Eventually, the assembler processing the assembly file next treats this line as a 16-bit operation. Hence, the binary code produced by the assembler and the CRL2 file analyzed by aiT would differ, leading to incorrect WCET estimates. In order to prevent this situation, the decision taken on the bit-width of an operation is passed to the assembler using particular assembler directives [6]:

```
.code32
MOV D9, D8
```

Loop Transformation

Second, the conversions from LLIR to CRL2 and back need to be aware of a control flow transformation inherently performed by aiT. As already mentioned previously, execution contexts provide higher precision of WCET estimates. Due to the structure of CRL2, context information can only be attached to CRL2 functions. However, context-related information is not only relevant for functions, but also for loops within functions. For example, it is useful to distinguish individual iterations of a loop during WCET analysis and to compute different WCET information for loop iterations, depending on e. g. cache states. This can only be achieved if contexts are also applicable to loops. This is done within aiT by moving loops out of their original function into a new dedicated CRL2 function which calls itself recursively during each iteration (cf. Figure 3). This way, aiT is able to attach context-related information to loops by simply annotating the newly created loop functions.

However, this loop transformation stage has the effect that the CFG structures of LLIR and CRL2 differ. Since a CRL2 CFG may contain more functions than its LLIR counterpart, there is no direct correspondence at the function level. When traversing the CRL2 CFG in order to import all WCET data computed by aiT into the WCC compiler, CRL2 functions may be reached for which no LLIR

[1] We could also assume a 16-bit operation – the actual width does not matter. However, it is important to feed the decision to the assembler.

function exists. Hence, the question arises where to store all the WCET data attached at CRL2 functions within LLIR.

Basic blocks have a unique identifier in both IRs, namely the label denoting its starting address. WCET information stored at the level of CRL2 functions is attached to the very first basic block of this function within LLIR. For CRL2 functions F also existing within LLIR, this has the effect that all WCET data of F will also be available for F within LLIR by just checking F's first basic block. For CRL2 functions F' generated during loop transformation, the WCET data attached to F' will be stored at that position within LLIR where the loop represented by F' begins. This way, all WCET information is stored exactly at those code positions within LLIR to which the WCET data belongs.

3.3. LLIR Objectives and Handlers

In a simple approach, all WCET data extracted from CRL2 would directly be attached to the corresponding LLIR components as described in the previous section, e. g. the class of LLIR basic blocks would be extended by new attributes storing the execution count and the block's WCET. However, this approach is too inflexible for our purposes.

As already shown in Section 2, WCET-related compiler optimizations will have to deal with trade-offs between WCET and other optimization criteria, e. g. code size [13]. In the future, we intend to extend the WCC infrastructure presented in this paper towards a full multi-objective optimization engine. Besides the WCET data, the compiler will thus have to deal with other types of information representing other optimization objectives. Hence, the simple approach to just store this additional data within the LLIR core classes like e. g. basic blocks in the future will require the entire LLIR core to be extended and rebuilt. But since a compiler's IR is a complex piece of software, modifications of its core should be reduced to an absolute minimum.

In order to be able to attach arbitrary data to the LLIR core components without having to modify the core in the future, we have extended the LLIR by a modular and flexible objective and handler mechanism.

An *LLIR objective* represents a container storing all data relevant for the compiler in order to perform optimizations for a particular objective. Besides its pure data elements, methods in order to set and get the data stored within an LLIR objective are provided. Additionally, each LLIR objective carries specific type information such that a WCET LLIR objective can be distinguished from e. g. a code size objective by simply comparing their type attributes.

Using inheritance from an abstract objective class, other kinds of LLIR objectives besides WCET can be created easily. Arbitrary types of LLIR objectives can now be attached to the entire LLIR control flow graph, to LLIR functions, basic blocks and instructions. All of these LLIR components contain a so-called *objective handler* which is responsible for managing multiple LLIR objectives of different type attached to the same LLIR component. In this way, arbitrary data useful for compiler optimizations pursuing different optimization goals can be freely attached to the LLIR. The objective handler takes care that only one instance of an LLIR objective with a given type can be attached to an LLIR component, so that all data relevant e. g. for the object's WCET is not scattered among various instances of the WCET LLIR objective. The objective handler provides

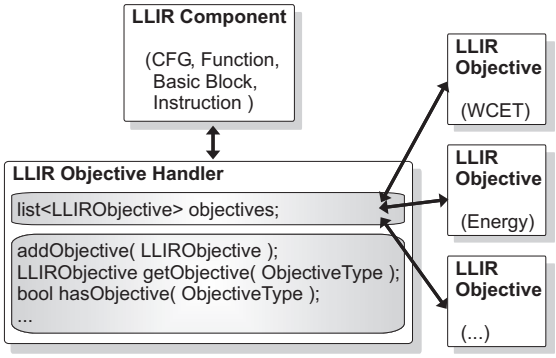


Figure 4. LLIR Objective Handling

methods in order to set and get objectives of a certain objective type. The structure of the LLIR objective and handler mechanism is illustrated in Figure 4.

4. Discussion of the Structure of WCC

The proposed architecture of our WCET-aware C compiler WCC has several advantages compared to previously published approaches (cf. Table 1).

First, the use of both a high- and a low-level IR is extremely beneficial for WCET analysis and optimization. WCET analysis inherently requires high-level flow facts to be present. Previously published approaches recompute these flow facts from a low-level IR. This approach is cumbersome and inelegant. The availability of a high-level IR within our WCC compiler results in an overall simplification of flow fact computation since flow facts are computed directly at the corresponding level of abstraction. In addition, there are several high-level control flow optimizations potentially having a positive effect on WCET. For example, procedure cloning making use of calling contexts is easier to realize using a high-level IR than using LLIR.

Second, our WCC compiler does not rely on a proprietarily developed WCET analyzer. Instead, we were able to tightly integrate aiT into our compiler. By coupling WCC with AbsInt's aiT, we do not use simplified models of the target processor architecture at all. In contrast to various other publications, no simplifying assumptions on e. g. pipeline behavior and cache structure are made during WCET analysis. This is ensured by the high quality of aiT where the entire processor architecture is modeled with a very high degree of accuracy. As a consequence, the WCET data our compiler uses is highly accurate and precise.

Due to the accuracy and precision of the processor models used by aiT, a vast amount of WCET-related information is computed. Since both the compiler and aiT use CRL2 as common exchange format, all this data is available within WCC without limitations. Currently, we are able to import the following data from aiT into our compiler: global WCET of an entire CFG, calling contexts, context-dependent WCETs and execution counts of basic blocks and functions, feasibility of CFG edges, overall WCETs of basic blocks. Using the infrastructure presented in this paper, it is easy to import all other WCET data from aiT to WCC like e. g. value ranges of registers or pipeline and cache states.

The concept of objectives and handlers presented in this paper allows to model arbitrary trade-offs within our compiler. Previous publications have already shown that it is relevant to combine WCET optimization with other criteria like e. g. code size. Using objectives and handlers, our

	High/Low-Level IR	Tight WCET Integration	Automated Flow Facts	Compiler Optimization Support	Complex Processor	Avail. WCET Data	Multi-Objective	Tested
WCC	+	+	±	±	+	+	+	+
[4]	–	+	–	–	–	–	–	–
[7]	+	–	–	+			–	–
[12]	–	–	+	+	–	–	–	+
[17, 18]	–	+	+	+	–	–	–	+
[13]	–	–			–	–	+	+
[16]	–	–	+	+	±	–	–	+

Table 1. Comparison of WCC with Related Work

compiler is already well designed for this purpose, even if this is part of our future work.

5. Summary and Future Work

This paper is the first one describing a tight integration of an existing WCET analyzer into a compiler infrastructure. Using our techniques, it is possible to feed the assembly-like contents of our compiler's low-level IR LLIR into AbsInt's aiT automatically. This is achieved by translating LLIR into aiT's intermediate format CRL2. Since LLIR is an assembly-level representation, some information only visible at the linker level (e. g. exact opcode formats of operations having the same mnemonic) is unavailable. Hence, care has been taken during the translation from LLIR to CRL2 that the generated CRL2 representation of a given program is fully equivalent to its LLIR representation, and that the same holds for the final executable program generated by assembling and linking the LLIR representation.

After CRL2 generation, the WCET analyzer aiT is invoked automatically by our compiler. After its timing analyses are completed, the WCET data produced by aiT is imported into LLIR for further compiler optimizations.

In order to integrate WCET data within LLIR, a novel mechanism for future multi-objective compiler optimization is employed. It allows to attach arbitrary data relevant for compiler optimizations pursuing different optimization goals to almost all components of the LLIR. For example, this objective handler technique enables to store code size or energy and WCET data within LLIR at the same time in a flexible, modular and extensible way. This multi-objective design of the LLIR is motivated by the fact that the design of low-cost real-time systems requires performing automated trade-offs between different optimization criteria.

Our compiler is equipped with an extensive testbench consisting of 1,500 C files with 165,000 lines of code. The WCC compiler successfully compiles this testbench, generates valid CRL2 and performs WCET computation using aiT in the background. These tests serve as a proof of concept for the techniques presented in this paper and clearly demonstrate the reliability and quality of our concepts.

As already stated in Section 2, a good WCET analysis framework relies on the presence of high-level flow facts. To make them available within LLIR, we need to compute these flow facts within our high-level IR ICD-C, keep them up to date during optimization and pass them through the code selector down to the LLIR. Thus, the integration of flow fact computation [8] and transformation [12] into our WCC is part of the future work. Hence, the corresponding entries for WCC in Table 1 are marked just with ±.

Of course, the main part of our future work is to develop WCET-aware compiler optimizations both at the high ICD-C level and at the LLIR level. This requires some kind of back annotation of the WCET data stored in the LLIR up to our high-level IR. Using our compiler structure with WCET data available at all levels, several issues noted in the WCET compiler wish list [3] can be tackled in the future.

Besides pure WCET-aware optimizations, we will consider multi-objective optimizations within our compiler in order to achieve trade-offs between real-time constraints and other optimization criteria.

Acknowledgments

The authors would like to thank Jens Wagner, Jörg Eckart and Luis Gomez who have designed the LLIR.

References

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. *http://www.absint.com/ait*, 2005.

[2] AbsInt Angewandte Informatik GmbH. CRL Version 2. *http://www.absint.com/artist2/doc/crl2*, 2006.

[3] G. Bernat and N. Holsti. Compiler Support for WCET Analysis: a Wish List. In *Proc. of "3rd Intl. Workshop on WCET Analysis" (WCET)*, Porto, July 2003.

[4] H. Börjesson. Incorporating Worst Case Execution Time in a Commercial C-Compiler. Master's thesis, Uppsala University, Jan. 1996.

[5] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley, Boston, 2001.

[6] D. Elsner and J. Fenlason. *Using as – The GNU Assembler*. Free Software Foundation, 1994.

[7] J. Engblom. Worst-Case Execution Time Analysis for Optimized Code. Master's thesis, Uppsala University, Uppsala, Sept. 1997.

[8] J. Gustafsson, A. Ermedahl and B. Lisper. Towards a Flow Analysis for Embedded System C Programs. In *Proc. of "The 10th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems" (WORDS)*, Sedona, Feb 2005.

[9] *TriCore 1 32-Bit Unified Processor Core v1.3 Architecture – Architecture Manual*. Infineon Technologies AG, Sept. 2002.

[10] Informatik Centrum Dortmund. ICD-C Compiler framework. *http://www.icd.de/es/icd-c*, 2006.

[11] *ICD Low Level Intermediate Representation backend infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2006.

[12] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. of "13th Euromicro Conference on Real-Time Systems" (ECRTS)*, Delft, Jun 2001.

[13] S. Lee, J. Lee, C. Y. Park and S. L. Min. A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor. In *Proc. of "Intl. Workshop on Software and Compilers for Embedded Systems" (SCOPES)*, Amsterdam, Sept. 2004.

[14] R. Leupers. *Code Optimization Techniqes for Embedded Processors - Methods, Algorithms and Tools*. Kluwer Academic Publishers, Boston, 2000.

[15] S. Steinke, L. Wehmeyer et al. The *encc* Compiler Homepage. *http://ls12-www.cs.uni-dortmund.de/research/encc*, 2002.

[16] L. Wehmeyer and P. Marwedel. Influence of Onchip Scratchpad Memories on WCET Prediction. In *Proc. of "4th Intl. Workshop on WCET Analysis" (WCET)*, Catania, June 2004.

[17] W. Zhao, W. Kreahling, D. Whalley et al. Improving WCET by Optimizing Worst-Case Paths. In *Proc. of "11th RTAS Symposium"*, San Francisco, Mar 2005.

[18] W. Zhao, P. Kulkarni, D. Whalley et al. Tuning the WCET of Embedded Applications. In *Proc. of "10th RTAS Symposium"*, Toronto, May 2004.

Loop Nest Splitting for WCET-Optimization and Predictability Improvement *

Heiko Falk Martin Schwarzer

University of Dortmund, Computer Science 12, D - 44221 Dortmund, Germany
Heiko.Falk | Martin.Schwarzer@udo.edu

Abstract

This paper presents the influence of the loop nest splitting source code optimization on the worst-case execution time (WCET). Loop nest splitting minimizes the number of executed if-statements in loop nests of embedded multimedia applications. It identifies iterations of a loop nest where all if-statements are satisfied and splits the loop nest such that if-statements are not executed at all for large parts of the loop nest's iteration space.

Especially loops and if-statements of high-level languages are an inherent source of unpredictability and loss of precision for WCET analysis. This is caused by the fact that it is difficult to obtain safe and tight worst-case estimates of an application's flow of control through these high-level constructs. In addition, the corresponding control flow redirections expressed at the assembly level reduce predictability even more due to the complex pipeline and branch prediction behavior of modern embedded processors.

The analysis techniques for loop nest splitting are based on precise mathematical models combined with genetic algorithms. On the one hand, these techniques achieve a significantly more homogeneous structure of the control flow. On the other hand, the precision of our analyses leads to the generation of very accurate high-level flow facts for loops and if-statements. The application of our implemented algorithms to three real-life multimedia benchmarks leads to average speed-ups by 25.0% – 30.1%, while WCET is reduced between 34.0% and 36.3%.

1. Introduction

In contrast to general-purpose systems, embedded systems often have to meet real-time constraints. The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced. Besides the criticality of safeness of real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing of a real-time application, the designer tends to use oversized hardware in order to guarantee the safeness of the real-time system.

In recent years, the real-time behavior of embedded multimedia applications (e. g. medical image processing, video compression) with simultaneous consideration of power efficiency has become a crucial issue. Many of these applications are data-dominated using large amounts of data memory. Typically, such applications consist of deeply nested *for*-loops. The main algorithm is usually located in the innermost loop. Often, such an algorithm treats particular parts of its data specifically, e. g. an image border requires other manipulations than its center. This boundary checking

*This work is partially funded by the European IST FP6 Network of Excellence ARTIST2.

```
for (x=0; x<36; x++) { x1=4*x;
 for (y=0; y<49; y++) { y1=4*y;   /* y loop */
  for (k=0; k<9; k++) { x2=x1+k-4;
   for (l=0; l<9; l++) { y2=y1+l-4;
    for (i=0; i<4; i++) { x3=x1+i; x4=x2+i;
     for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
      if (x3<0 || 35<x3 || y3<0 || 48<y3)
         then_block_1; else else_block_1;
      if (x4<0 || 35<x4 || y4<0 || 48<y4)
         then_block_2; else else_block_2; }}}}}}
```

Figure 1. A typical Loop Nest (from MPEG 4)

is implemented using *if*-statements in the innermost loop (see e. g. Figure 1, an MPEG 4 full search motion estimation kernel [9]).

This code fragment has several properties making it suboptimal w. r. t. worst- and average-case execution time *(ACET)*. First, the *if*-statements lead to a very irregular control flow. Any jump instruction in a machine program causes a control hazard for pipelined processors [13]. This means that the pipeline needs to be stalled for some instruction cycles, so as to prevent the execution of incorrectly prefetched instructions. WCET analysis is faced with the problem to estimate whether a jump is taken or not. The worst-case influence of this decision on pipeline and branch prediction behavior needs to be taken into account. Since it is very difficult to predict these control flow modifications accurately, resulting WCETs tend to become imprecise the more irregular the control flow is.

In addition, the way how conditions are expressed within *if*-statements may also have a negative impact on WCET. If conditions are connected using the logical and / or operators of ANSI-C [10], they are evaluated lazily. For example, expression `e2` of the condition `e1 || e2` is not evaluated if `e1` already evaluates to true. Hence, each occurrence of the `||` and `&&` operators implies hidden control flow modifications having a negative influence on WCET. This source of unpredictability caused by the *if*-statements becomes even more severe if the *if*-statements are located in the heart of a loop nest as depicted in Figure 1. Here, WCET analysis has to multiply the overestimated data computed for the *if*-statements with the possibly also overestimated number of loop iterations, leading to even more imprecise estimates.

Considering the example shown in Figure 1, loop nest splitting is able to detect that

- the conditions `x3<0` and `y3<0` are never true,
- both *if*-statements are true for `x` ≥ 10 or `y` ≥ 14.

Information of the first type is used to detect conditions not having any influence on the control flow of an application. This kind of redundant code (which is not typical dead code, since the results of these conditions are used within the *if*-statement) can be removed from the code, thus reducing sources of unpredictability during WCET analysis of a program.

```
for (x=0; x<36; x++) { x1=4*x;
 for (y=0; y<49; y++)
  if (x>=10 || y>=14)        /* Splitting-If */
   for (; y<49; y++)        /* Second y loop */
    for (k=0; k<9; k++)
     ...              /* l- & i-loop omitted */
       for (j=0; j<4; j++) {
        then_block_1; then_block_2; }
  else { y1=4*y;
   for (k=0; k<9; k++) { x2=x1+k-4;
    ...              /* l- & i-loop omitted */
      for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
       if (0 || 35<x3 || 0 || 48<y3)
          then_block_1; else else_block_1;
       if (x4<0 || 35<x4 || y4<0 || 48<y4)
          then_block_2; else else_block_2; }}}}}
```

Figure 2. Loop Nest after Splitting

Using the second information, the entire loop nest can be rewritten so that the total number of executed *if*-statements is minimized (see Figure 2). In order to achieve this, a new *if*-statement (the *splitting-if*) is inserted in the y loop testing the condition $x \geq 10$ || $y \geq 14$. The *else*-part of this new *if*-statement is an exact copy of the body of the original y loop shown in Figure 1. Since all *if*-statements are fulfilled when the splitting-if is true, the *then*-part consists of the body of the y loop without any *if*-statements and associated *else*-blocks. To minimize executions of the splitting-if for values of $y \geq 14$, a second y loop is inserted in the *then*-part counting from the current value of y to the upper bound 48. The correctly transformed code is illustrated in Figure 2.

As shown by this example, our technique is able to generate a very homogeneous control flow in the hot-spots of an application. Furthermore, the algorithms briefly summarized in this paper enable the generation of precise high-level flow facts for WCET analysis. This paper evaluates the effect of loop nest splitting on the WCET of selected real-life benchmarks. Loop nest splitting is done by automatically transforming ANSI-C source codes. These source codes are then compiled for the ARM7 processor. WCET analysis for the resulting executable programs is finally done using the aiT WCET analyzer.

Section 2 of this paper gives a survey of related work. Section 3 presents the analyses and optimizations of of loop nest splitting. Section 4 describes the benchmarking results, and Section 5 summarizes and concludes this paper.

2. Related Work

Loop transformations have been described in literature on compiler design for many years (see e. g. [2, 13]) and are often integrated into today's optimizing compilers. Classical *loop splitting* (or *loop distribution* / *fission*) creates several loops out of an original one and distributes the statements of the original loop body among all new loops. The main goal of this optimization is to enable the parallelization of a loop due to fewer data dependencies [2] and to possibly improve I-cache performance due to smaller loop bodies. The impact of this optimization on WCET has not yet been studied.

Loop unswitching is applied to loops containing loop-invariant *if*-statements [13]. The loop is then replicated inside each branch of the *if*-statement, reducing the branching overhead and decreasing code sizes of the loops [2]. The goals of loop unswitching and the way how the optimization is expressed are equivalent to the topics of Section 1. But the fact that the *if*-statements must not depend on index variables makes loop unswitching unsuitable for multimedia programs. The fact that only loop-invariant conditions are considered implies that no valuable flow facts for WCET analysis are generated during this optimization.

In [12], classical loop splitting is applied together with function call insertion at the source code level to improve I-cache performance. After the application of loop splitting, a large reduction of I-cache misses is reported for one benchmark. All other parameters (instruction and data memory accesses, D-cache misses) are worse after the transformation. All results are generated with cache simulation software which is known to be imprecise, and the WCETs and ACETs of the benchmark are not considered at all.

This survey of related work shows that loop optimizations typically aim at improving temporal or spatial locality of caches and thus focus on ACET reduction. The influence of loop optimizations on WCET has not yet been studied thoroughly. Loop nest splitting was originally presented in [6, 8]. However, these original publications solely concentrated on the optimization of average-case execution time and energy dissipation. The impact of the optimization on WCET was not yet taken into account. Furthermore, all control-flow related data computed during the optimization process was discarded after loop nest splitting and was not used during subsequent optimization or analysis steps.

3. Analysis and Optimization Algorithm

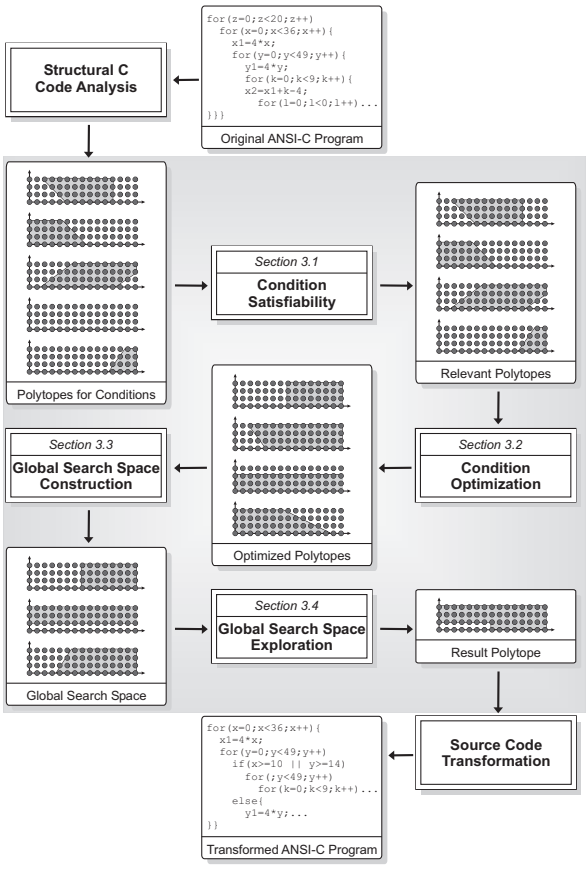


Figure 3. Design Flow of Loop Nest Splitting

Figure 3 gives an overview over the techniques required for loop nest splitting. As can be seen from this figure, loop nest splitting relies on polyhedral models in order to represent loop nests and *if*-statements. Polyhedra and

polytopes are defined as follows:

Definition:

1. $P = \{x \in \mathbb{Z}^N \mid Ax = a,\ Bx \geq b\}$ is called a *polyhedron* for $A, B \in \mathbb{Z}^{m \times N}$ and $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$.
2. A polyhedron P is called a *polytope*, if $|P| < \infty$.

Since polyhedra are systems of linear inequations, loop nest splitting requires loop bounds and conditions of *if*-statements to be affine expressions depending on the loops' index variables. For a given loop nest $\Lambda = \{L_1, \ldots, L_N\}$ where each loop L_l is characterized by its index variable i_l and lower / upper bounds lb_l and ub_l, loop nest splitting computes values lb'_l and ub'_l for every loop $L_l \in \Lambda$ with

- $lb'_l \geq lb_l$ and $ub'_l \leq ub_l$, i. e. the computed values must lie within the loop bounds,
- all loop-variant *if*-statements in Λ are satisfied for all values of the index variables i_l with $lb'_l \leq i_l \leq ub'_l$,
- loop nest splitting by all values lb'_l and ub'_l leads to the minimization of *if*-statement execution.

The values lb'_l and ub'_l are used for the construction of the splitting *if*-statement. The individual steps carried out during loop nest splitting as shown in Figure 3 are briefly described in Section 3.1 (cf. also [7] for a more in-depth description). Section 3.2 deals with the automatic generation of high-level flow facts during loop nest splitting.

3.1. Workflow of Loop Nest Splitting

Since the analyses of loop nest splitting require that the source code to be optimized meets some preconditions, these requirements are checked in the very beginning. During this phase labeled "Structural C Code Analysis" in Figure 3, only suitable loop nests and *if*-statements having affine bounds and conditions are extracted from the source code. The output of this phase consists of a set of polytopes, each of them representing a single condition occurring in the source code. The core optimization algorithm consists of four sequentially executed tasks that are illustrated as a shaded region in Figure 3. In the beginning, all conditions in a loop nest are analyzed separately without considering any inter-dependencies among them.

First, it is detected if conditions ever evaluate to true or not ("Condition Satisfiability"). For example, the two conditions `x3<0` and `y3<0` are eliminated from the code shown in Figure 1, since they are provably false during each loop nest iteration and are thus represented by empty polytopes.

Second, all satisfiable conditions are analyzed and an optimized search space for each condition is constructed ("Condition Optimization"). This means, that a polytope P representing an original condition C is replaced by an optimized polytope P' modeling a condition C' such that $C' \Rightarrow C$ holds. The goal is to generate P' in such a way that C' is significantly simpler than C. For example, condition optimization detects that $C' =$ `x >= 10` implies $C =$ `4*x + k + i >= 40` for the loops of Figure 1.

In a third step, all polytopes P' generated during condition optimization are combined to form a global search space G ("Global Search Space Construction"). This stage is motivated by the fact that the previous phases only considered single conditions of entire *if*-statements in isolation. In order to determine value ranges of the loop index variables, for which all *if*-statements in a loop nest are true, all P' need to be combined using intersection and union of polytopes according to the structure of all *if*-statements.

Finally, this global search space G has to be explored leading to the optimized result for loop nest splitting ("Global Search Space Exploration"). Basically, this phase selects a subset of constraints defining the global search space G in order to build a final polytope R representing the splitting *if*-statement. For the code shown in Figure 1, the outcome of the global search space exploration is the polytope $R = \{\mathtt{x} \geq 10\} \cup \{\mathtt{y} \geq 14\}$.

The result R of global search space exploration is finally used to rewrite a loop nest ("Source Code Transformation"). For this purpose, the splitting *if*-statement has to be generated and inserted in the loop nest. Its *then*- and *else*-parts are created by replicating parts of the original loop nest.

3.2. Flow Fact Generation

Since the major part of the execution time of a program is spent in loops, the iteration counts play an important role for WCET estimation. Hence, it is crucial to pass precise information about the number of loop iterations to the WCET analyzer in order to obtain safe and accurate WCET bounds. As already stated in the previous section, polytopes are used to model conditions and loops. Since polytopes are represented by linear inequations, the bounds of a loop L_l necessarily have to be affine expressions of the surrounding loops for loop nest splitting. An outermost loop L_1 is not surrounded by any other loop so that its bounds lb_1 and ub_1 are required to be constant. This way, it is ensured that the loop's iterations are allowed to be non-constant but still are fully analyzable at compile time.

During loop nest splitting, polytopes are generated modeling the loop currently under analysis. This is done straightforward by defining affine constraints for the lower and upper bounds of the loop itself and for all surrounding loops. In the resulting polytope, each integral point represents a single execution of the current loop body for one actual assignment of values to the loops' index variables. By counting the number of integral points of these polytopes, the total number of executions of the loop body can be determined exactly. For this purpose, so-called *Ehrhart polynomials* [4] are applied to the polytopes.

Example: *For the* `l`*-loop shown in Figure 1, a polytope having the constraints* `0 <= x < 36`, `0 <= y < 49`, `0 <= k < 9` *and* `0 <= l < 9` *would be generated. This polytope contains 142,884 points. Hence, the body of the* `l`*-loop is executed as many times within the entire loop nest.*

This number of integral points is used to generate flow facts for WCET analysis that exactly specify the number of executions of a loop body compared to the code lying outside the outermost loop L_1. In the case of the aiT WCET analyzer [1] used throughout this work, annotations like `flow 0x40007c / 0x40002e is exactly 142884;` are created, where the given addresses represent the basic blocks lying inside the current loop and outside the outermost loop, respectively.

In addition, more annotations concerning the splitting-if generated after global search space exploration can be provided to aiT. As already mentioned in Section 3.1, the final solution of loop nest splitting is a polytope R which is used to generate the splitting-if. The computation of the size of R using Ehrhart polynomials leads to the actual number of

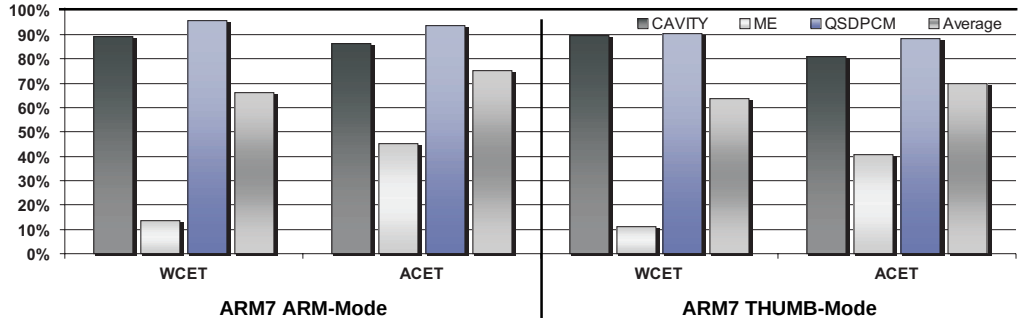


Figure 4. Relative WCETs and ACETs after Loop Nest Splitting

loop iterations for which the splitting-if provably is true. Since all loops are fully analyzable at compile time, the number of iterations for which the splitting-if is not true can also be computed. These two values are used to generate additional `flow` annotations for aiT precisely modeling the control flow structures resulting from loop nest splitting.

4. Evaluation

In this section, the impact of loop nest splitting on WCET is evaluated. For this purpose, the benchmarking workflow is presented in Section 4.1. Benchmarking results are given in Section 4.2.

4.1. Benchmarking Methodology

The techniques presented in Section 3 are fully implemented using the SUIF [16], Polylib [15] and PGAPack [11] libraries. Both GAs use the default parameters provided by [11] (population size 100, replacement fraction 50%, 1,000 iterations). Our tool was applied to three multimedia programs. First, a medical tomography image processor (*CAVITY* [3]) is used. The second benchmark is an MPEG 4 full search motion estimation (*ME* [9], see Section 1), and the QSDPCM algorithm [14] for scene adaptive coding serves as third benchmark.

Since all polyhedral operations used [15] have exponential worst case complexity, loop nest splitting as a whole also has exponential complexity. Nevertheless, the effective runtimes of our tool are very low, only a maximum of 1.58 CPU seconds (CAVITY) are required for optimization on an AMD Athlon running at 1.3 GHz.

In order to quantify the influence of loop nest splitting on the WCET of the benchmarks, we considered an ARM7 based processor architecture. The ARM7 is a dual instruction set CPU having a 32-bit wide ARM instruction set and a 16-bit THUMB instruction set. For both instruction sets, the native ARM compilers *armcc* and *tcc* were used to generate executable code from the benchmark's source codes. Both compilers are always invoked with all optimizations enabled so that highly optimized code is generated.

In a first step, the source codes were compiled for both instruction sets without loop nest splitting being applied. The resulting executables were passed to AbsInt's WCET analyzer aiT [1] for the ARM7 to obtain the WCETs before our optimization. In addition to the binary executable, a specification file containing the exact number of loop iterations is also provided to aiT. In parallel, the same executables were processed by the cycle-true native ARM simulator. These simulations used typical input data for the benchmarks and the resulting cycle counts are considered as the ACETs of the benchmarks in the following.

In a second step, the source codes were optimized using our tool for loop nest splitting. The resulting optimized source codes were processed in the same way as described in the previous paragraph, leading to the corresponding WCETs and ACETs after loop nest splitting. For WCET analysis of the optimized codes, a specification file containing the `flow` annotations for loop bodies and the splitting-if (cf. Section 3.2) is also provided.

4.2. Benchmarking Results

WCET and ACET

Figure 4 shows the effects of loop nest splitting on the WCET and ACET of the benchmarks for the ARM7 using both the ARM and the THUMB instruction sets. The figure shows the corresponding values for the optimized benchmarks as a percentage of the unoptimized versions denoted as 100%.

As can be seen from this figure, loop nest splitting is able to reduce both ACET and WCET significantly. Concerning ACET, improvements between 6.4% (QSDPCM) and 54.8% (ME) were measured for the ARM instruction set. Similarly, ACET is reduced between 11.5% (QSDPCM) and up to 59.4% (ME) using THUMB instructions. On average for all considered benchmarks, ACET is reduced between 25.0% (ARM) and 30.1% (THUMB). These numbers clearly demonstrate that the generation of a homogeneous control flow within loop nests leads to increased average performance since a large amount of code located in the innermost loops before our optimization is eliminated.

However, Figure 4 also shows that the WCET reductions achieved after loop nest splitting have a similar order of magnitude. Here, the gains reach from 4.4% (QSDPCM) up to 86.5% (ME) when using 32-bit wide instructions. For the 16-bit THUMB instruction set, reductions of WCET between 9.6% (QSDPCM) and even 89.0% (ME) were reported by aiT. On average over all benchmarks, the reductions of WCET achieved by loop nest splitting are significantly larger than the corresponding ACET reductions. In terms of WCET, average improvements of 34.0% (ARM) and 36.3% (THUMB) can be reported.

Despite the fact that WCET is reduced more than ACET, Figure 4 does not show an invalid WCET underestimation. This is due to the fact that Figure 4 presents all results just as a percentage of the WCETs and ACETs of the unoptimized benchmarks. This way, it is legal to reduce WCET by 89% and ACET by just 59%. For all results presented in this paper, the estimated absolute WCETs are correct and safe and are larger than the corresponding absolute ACETs.

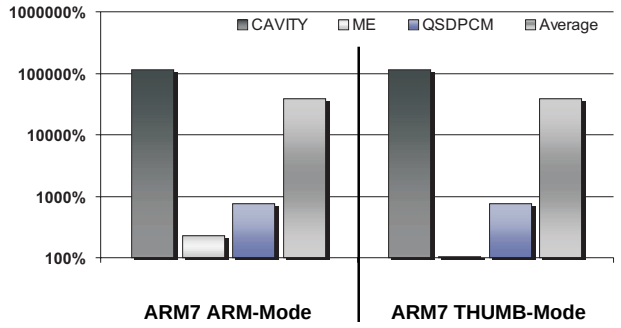


Figure 5. Relative WCETs after Loop Nest Splitting without Flow Annotations

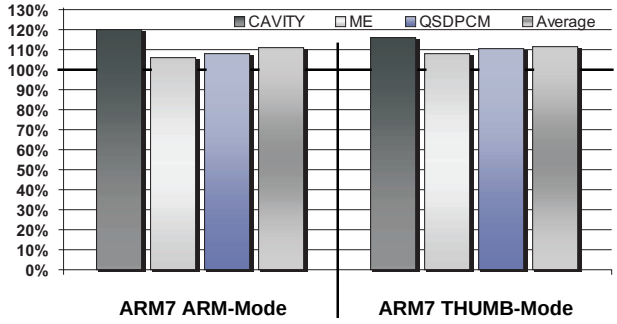


Figure 6. Relative Code Sizes after Loop Nest Splitting

Of course, these differences in the average WCET and ACET values are caused by the enormous improvements of WCET for the ME benchmark. For the two other benchmarks (CAVITY and QSDPCM), the WCET reduction scales with the corresponding ACET improvements. This behavior shows that the achievable gains in terms of WCET also depend on the overall structure of the unoptimized benchmark's code.

For example, both ME and QSDPCM have a similar structure like that shown in Figure 1. The difference between both benchmarks leading to the varying WCET reductions is the structure of the code blocks executed conditionally by virtue of the *if*-statements. For ME, `then_block_1` and `then_block_2` just contain the assignment of a constant to a variable, whereas both `else`-blocks contain very complex array accesses and address computations. Since these address computations invoke integer divisions and modulo computations, this code leads to the generation of calls to runtime libraries. For QSDPCM, the situation is vice versa – here, the *then*-blocks are more complex than the *else*-blocks.

This slight difference has the effect that for the unoptimized ME, the WCET path passes through the *else*-parts of the *if*-statements, whereas it lies on the *then*-parts for QSDPCM. After loop nest splitting, the new WCET path traverses the *then*-part of the splitting-if for both benchmarks. For ME, the innermost loop of this *then*-part now just contains the assignments of constants. Thus, the new WCET path of ME after loop nest splitting does no longer contain the costly address computations mentioned above, leading to the very high gains reported in this section. In the case of QSDPCM, the innermost loop of the *then*-part of the splitting-if still contains the complex address computations after loop nest splitting. As a consequence, this complex code still lies on the WCET path so that the gains in terms of WCET are not as high as compared to ME.

Impact of Flow Facts on WCET

The benefits of the flow facts extracted during loop nest splitting for WCET analysis are depicted in Figure 5. This diagram shows the WCETs resulting from the analysis of the benchmarks after loop nest splitting, but without providing aiT with the flow annotations precisely describing the splitting-if (cf. Section 3.2). Results are presented in a relative way such that the 100% baseline represents the WCETs before loop nest splitting.

As can be seen, the flow facts computed during global search space exploration are essential for a successful WCET minimization after loop nest splitting. Without this information, aiT is unable to compute precise WCETs from the optimized control flow structures. For all benchmarks, the WCETs without flow facts are worse than WCETs before any optimization being applied. For the ME benchmark, the degradations of WCET range between 4% (THUMB) and 231% (ARM). For QSDPCM, WCET results without flow facts are even worse. Here, increases between 743% and 767% compared to the WCETs of the unoptimized benchmarks were measured. The highest WCETs were computed for CAVITY. For this benchmark, WCETs increase between 113,031% (ARM) and 113,953% (THUMB) if aiT is not provided with the flow facts generated by loop nest splitting.

Code Size

Since code is replicated, loop nest splitting obviously entails a certain increase in code size that we do not want to neglect. However, Figure 6 shows that these increases are within small bounds. In order to measure code sizes, the size of the text sections in bytes was extracted from the ELF binaries of the benchmarks before and after loop nest splitting. For CAVITY, code size increases range between 19.9% (ARM) and 15.9% (THUMB). Although the ME benchmark is accelerated most, its code enlarges least. Increases of just 5.8% (ARM) and 8.1% (THUMB) were measured. Finally, the code of QSDPCM enlarges between 7.9% (ARM) and 10.5% (THUMB). On average over all benchmarks, code size increases of just 11.2% (ARM) and 11.5% (THUMB) were measured.

For fine tuned embedded systems with hard constraints on both worst-case execution time and code size, code size increases might potentially be a severe drawback. However, loop nest splitting offers inherent opportunities for solving this problem since it is perfectly suited for trading off WCET with code size increases.

As depicted in Figure 2, loop nest splitting generates a splitting-if like `if (x >= 10 || y >= 14)` and places it in the `y`-loop, since this is the innermost loop of the entire loop nest the splitting-if directly depends on. Within the splitting-if, the remaining loop nest consisting of the `k`-, `l`-, `i`- and `j`-loop can be found. Since the splitting-if does not depend on index variables of this remaining loop nest by definition, it is always legal to place the splitting-if in any of these loops. This way, the portions of code replicated during loop nest splitting become smaller on the one hand.

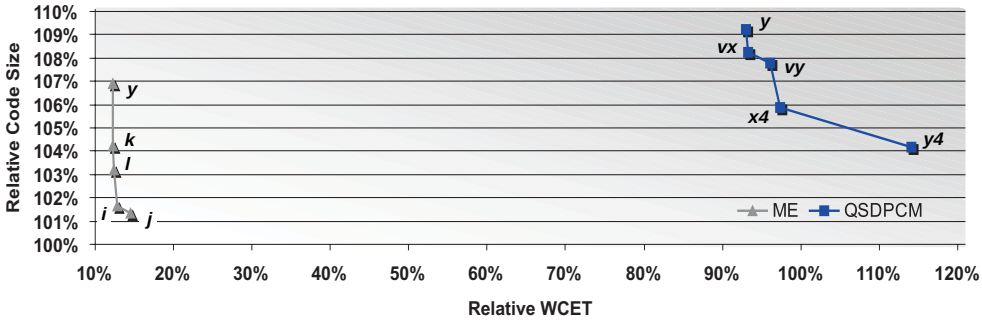


Figure 7. Possible WCET / Code Size Trade-Offs for Loop Nest Splitting

On the other hand, it can be expected that less improvements of WCET will be achieved since more *if*-statements are executed, leading to the mentioned trade-off.

Figure 7 shows the corresponding Pareto curves for the ME and QSDPCM benchmarks. The x-axis denotes the relative WCETs of the benchmarks, whereas the y-axis shows the corresponding relative code sizes (100% = unoptimized code version). Each point is labeled with the loop in which the splitting-if is placed. The code versions used to generate all previously presented results are marked with `y` (ME and QSDPCM). As expected, they lead to the lowest WCETs and entail the highest code size increases. In contrast, code versions `j` (ME) and `y4` (QSDPCM) are the slowest but smallest ones. In between these two extremal points, other interesting solutions for loop nest splitting can be found.

These experiments show that it is worthwhile to study possible trade-offs when applying loop nest splitting under tight code size constraints. A more systematic study than that presented here resulting in an automated approach to explore WCET / size trade-offs is part of the future work.

5. Conclusions

This paper puts the previously presented source code optimization loop nest splitting in the context of WCET. Loop nest splitting removes redundancies in the control flow of embedded multimedia applications. Using polytope models, conditions having no effect on the control flow are removed. Genetic algorithms identify ranges of the iteration space where all *if*-statements are provably satisfied. The source code of an application is rewritten in such a way that the total number of executed *if*-statements is minimized.

It has turned out that loop nest splitting is highly beneficial for WCET optimization. This is due to the fact that the quality of WCET analysis inherently depends on a precise description of the control flow of an application under analysis. On the one hand, precise high-level flow facts representing e. g. loop iterations have to be provided. On the other hand, assembly-level jumps modifying the control flow are hard to analyze since the conditions under which a jump is taken or not are difficult to analyze resulting in imprecise worst-case assumptions.

The benefits of loop nest splitting on WCET are twofold. First, the optimization by itself produces a very linear and homogeneous control flow in the hot-spots of an application. As a consequence, the potential for applying the imprecise worst-case assumptions mentioned above during WCET analysis of the time-critical parts of a code is heavily reduced. Second, loop nest splitting inherently computes execution frequencies of all relevant control flow constructs during its analyses. These execution frequencies can directly be used to formulate precise loop and flow annotations for the WCET analyzer.

The results presented in this paper underline the effectiveness of loop nest splitting. In terms of average-case execution times, it achieves improvements between 25.0% – 30.1%. However, even larger average gains are reported in terms of WCET. Here, reductions between 34.0% and 36.3% were measured for an ARM7 based processor.

In the future, we intend to integrate loop nest splitting into our WCET-aware C compiler [5]. Due to its multi-objective capabilities, it is perfectly suited to systematically explore the WCET / size trade-offs of loop nest splitting.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework.

References

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. *http://www.absint.com/ait*, 2005.

[2] D. F. Bacon, S. L. Graham et al. Compiler Transformations for High-Performance Computing. *ACM Computing Surv.*, 26(4), 1994.

[3] M. Bister, Y. Taeymans et al. Automatic Segmentation of Cardiac MR Images. *IEEE Journ. on Computers in Cardiology*, 1989.

[4] P. Clauss and V. Loechner. Parametric Analysis of polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, 19(2), July 1998.

[5] H. Falk and P. Lokuciejewski. Design of a WCET-Aware C Compiler. In *Proc. of "6th Intl. Workshop on WCET Analysis" (WCET)*, Dresden, July 2006.

[6] H. Falk and P. Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *Proc. of DATE*, Munich, Mar. 2003.

[7] H. Falk and P. Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Boston, Oct. 2004.

[8] H. Falk and M. Verma. Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In *Proc. of SCOPES*, Amsterdam, Sept. 2004.

[9] S. Gupta, M. Miranda et al. Analysis of High-level Address Code Transformations for Programmable Processors. In *Proc. of DATE*, Paris, 2000.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[11] D. Levine. *Users Guide to the PGAPack Parallel Genetic Algorithm Library*. Tech. Rep. ANL-95/18, Argonne National Lab., 1996.

[12] N. Liveris, N. D. Zervas et al. A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications. In *Proc. of DATE*, Paris, 2002.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[14] P. Strobach. A new technique in scene adaptive coding. In *Proc. of EUSIPCO*, Grenoble, 1988.

[15] D. K. Wilde. *A Library for doing polyhedral Operations*. Tech. Rep. 785, IRISA Rennes, France, 1993.

[16] R. Wilson, R. French et al. An Overview of the SUIF Compiler System. *http://suif.stanford.edu/suif/suif1*, 1995.

Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis

D. Kebbal and P. Sainrat
Institut de Recherche en Informatique de Toulouse
118 route de Narbonne - F-31062 Toulouse Cedex 9 France

Abstract

This paper examines the problem of determining bounds on execution time of real-time programs. Execution time estimation is generally useful in real-time software verification phase, but may be used in other phases of the design and execution of real-time programs (scheduling, automatic parallelizing, etc.). This paper is devoted to the worst-case execution time (WCET) analysis. We present a static WCET analysis approach aimed to automatically extract flow information used in WCET estimate computing. The approach combines symbolic execution and path enumeration. The main idea is to avoid unfolding loops performed by symbolic execution-based approaches while providing tight and safe WCET estimate.

1. Introduction

Real-time systems are systems in which the execution time is subject to some constraints, which may lead to undesirable consequences when they are not respected, especially in hard real-time systems. The constraint validation process requires the knowledge of the execution time or bounds on the execution time of programs. WCET analysis is a popular approach used in the temporal constraints validation of hard real-time systems.

Static WCET analysis performs a high-level static analysis of the program source or object code. This avoids working on the program input data. Static WCET analysis consists of determining an upper bound on the program execution time. For each component of the program (block, task, etc.), an upper bound on the time of its execution is estimated. This definition implies that WCET analysis is only able to provide upper bounds on WCET values rather than exact values. Therefore, WCET analysis must guarantee two main properties in order to keep real-time systems predictable and their cost financially reasonable: *Safeness*; and *Tightness of provided WCET values*.

Static WCET analysis proceeds generally in three phases [4, 7]: flow analysis, low-level analysis and WCET estimate computing. Flow analysis characterizes the execution sequences of the program's components and their execution frequency (execution paths). Generally, two types of flow information are extracted. The first category is related to the program structure and may be extracted automatically. The second category is related to the program functionality and semantics. This includes information about loop bounds and feasible/infeasible paths especially. This type of flow information is complex to automate and therefore is generally provided by the programmer as annotations [7, 2]. Low-level analysis evaluates the execution time of each program component on the target hardware architecture. The calculation phase uses the results of the two previous steps to compute a WCET estimate for the program.

The remainder of this paper is organized as follows: the next section presents the related work. In section 3, we introduce some concepts which will be used by the flow analysis approach. Section 4 describes and discusses the proposed block-based symbolic execution method. Finally, we conclude the paper and present some perspective issues.

2. Related work

One of the most popular methods for static WCET analysis are based on path analysis. Path-based approaches proceed by explicitly enumerating the set of the program execution paths [10, 1]. [9] describes a method based on cycle-level symbolic execution to predict the WCET of real-time programs on high performance processors. The main drawbacks of those approaches lie in the important number of the generated program paths which scales exponentially with the program size. Another category of approaches called IPET[1] do not enumerate all program paths, but rather consider that they implicitly belong to the problem solution. The problem of the WCET estimation may then be con-

[1] Implicit Path Enumeration Techniques.

verted to the one of solving an ILP[2] problem [7, 11, 4].

All those approaches involve the programmer in the flow information determination process, especially the flow information related to program semantics (feasible/infeasible paths, loop bounds, etc.). Though the provided flow information may be highly precise, this is an error-prone problem. [5] uses an interval-based abstract interpretation method that associates ranges to the program variables and allows to automatically extract flow information related to program semantics. The method proceeds by rolling out the program (especially loops) until it terminates, which is very costly in time and memory. [8] presents an approach for automatic parametric WCET analysis. The method uses abstract interpretation, a symbolic method to count integer points in polyhedra and a symbolic ILP technique. The approach seems complex in practice. In [6], an approach for determining loop bounds is presented. They consider loops with multiple exit conditions and non-rectangular loops, in which the number of iterations of an inner loop depends on the current iteration of an outer loop. However, they handle only loops with the induction variable being increased by a constant amount between two successive iterations. Moreover, only the flow information related to loop bounds is determined. Symbolic execution is another technique for automatically extracting the flow information related to program functionality. The program is rolled out which allows to determine the values of variables as expressions of the program inputs [3]. Symbolic execution-based methods are capable to work with small programs, but are not well suited for long and complex programs.

Our aim is to determine an approach which automatically extracts flow information related to program functionality and computes a safe and tight upper bound on the program WCET with a lower cost. We use a hybrid method based on symbolic execution and path enumeration. Loops are not unfolded, rather a path analysis is performed on each loop block.

3. Flow analysis concepts

In the following, we present a set of concepts used by our flow analysis approach.

3.1. Program representation

We use the control flow graph (CFG) formalism to express the control flow of the program to be analyzed. The source code of the program is decomposed into a set of basic blocks. A basic block is a set of instructions with a single entry point and a single exit point. The entry point is situated at the beginning of the block and the exit point at its end. Two fictitious blocks, labeled $start$ and $exit$ are added. We assume that all executions of the CFG start at the $start$ block and end at the $exit$ block. Figure 1-b illustrates an example of a control flow graph of a program where the C source code is shown in figure 1-a. Formally, the program is represented by the graph $G = (B, E)$, where B represents the program basic blocks and E the precedence constraints between them.

[2] Integer Linear Programming.

3.2. Blocks and block graphs

We use the notion of block where a set of blocks of level l are grouped into a block of level $l-1$. Complex blocks correspond to complex programming language features (loops, conditional statements, functions, modules, etc.). The block composition starts at the lowest level and may be recursively carried out until the CFG level. Figure 2 illustrates the block graphs constructed for the example of the figure 1. Formally, a block b of level l is defined by the formula 1 and composed of: a number of sub-blocks (B_b); a set of header blocks ($B_b^h \subseteq B$, $B_b^h \subseteq B_b$); a set E_b of edges connecting the sub-blocks; and one or more exit edges ($E_b^e \subseteq E_b$).

$$b = \{B_b, B_b^h, E_b, E_b^e\}. \quad (1)$$

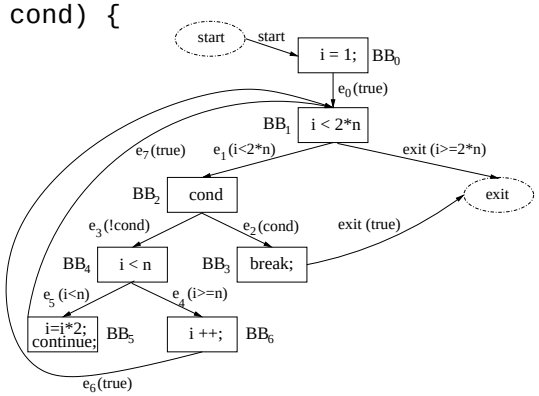
Each block b of level l is defined by a *block graph* describing its structure. The b's blocks set B_b is composed of blocks of higher levels ($m > l$). The set of edges E_b is constructed as follows: each edge of the CFG connecting two nodes belonging to two different blocks b_i and b_j of B_b forms an edge of level l from b_i to b_j. Edges to blocks outside B_b produce edges of *exit* type. Redundant edges are eliminated. In figure 2, in the graph of block b_0^1 corresponding to the *while* loop, the edge e_4 connecting the basic blocks BB_4 and BB_6 in the CFG yields the edge e_4^2.

A header block is a basic block executed when the execution flow reaches the block for the first time. Informally, header blocks correspond to loop and selection condition test blocks. The set of header-blocks of the block b_0^1 is $B_b^h = \{BB_1\}$ (figure 2). We handle only well-structured code programs yielding blocks with one header block. When the execution of the block is terminated, the control flow leaves the block through an exit-edge. The set of exit-edges of the block b_0^1 is $\{e_1^2, exit\}$. When the execution of a block must be repeated, this is done by transferring the execution flow to the header block through a back-edge. The set of back-edges of the block b_0^1 is $\{e_3^2, e_5^2\}$ (figure 2).

3.3. Paths and iterations

A path in a block graph b is a sequence of edges in E_b where the end-node of each edge is the starting-node of the next edge in the path. In the following we refer by *block-paths* to the *one-iteration paths* in a block b. We distinguish

```
void some_func(int n, int cond) {
  int i = 1;
  while (i < 2*n) {
    if(cond)
      break;
    ... //statements
    if(i < n) {
      i = 2*i;
      continue;
    }
    i++;
  }
}
```



a) C source code. b) Control flow graph.

Figure 1. Example program.

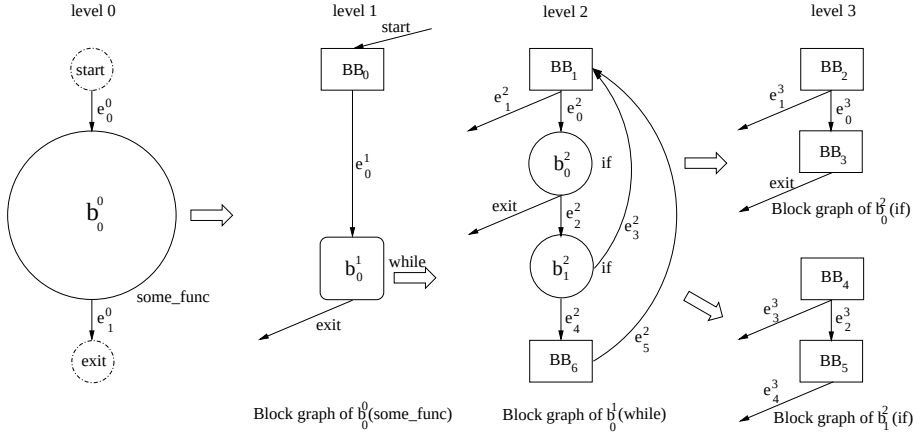


Figure 2. The block graphs of the example.

two types of block-paths: *exit_paths* and *loop_paths*. An exit-path in b is a path starting at the header block of b and ending by an exit-edge of b. Likewise, a loop-path in b is a path starting at the header block of b and ending by a back-edge of b. In figure 2 and table 1, the block b_0^1 has two exit paths: p_0^2 and p_1^2, and two loop-paths: p_2^2 and p_3^2.

Each block is executed a number of times (0 or more). We use the notion of *iteration* to denote an execution of a block, which is defined as one execution of a block-path.

4. Flow analysis approach

In this section, we describe our method aimed to automatically extract the flow information related to program semantics. We use a data flow analysis approach in order to derive values of variables at different points in the program. The approach combines symbolic execution with path enumeration. The flow analysis is performed for each block without unfolding iterative blocks. Rather than, the number of times the blocks are executed is analytically computed which reduces the complexity of the method. Only a subset of the symbolic states set of the program are computed (states at the entry point and the exit points of the block). Exit points of a block are the starting points of its exit edges.

A symbolic execution state may be represented by a triple $< V, PC, IP >$, where: $V = \{< v_1, e_1 >, \ldots, < v_n, e_n >\}$ is the set of pairs $< variable, expression >$, where the variables $v_1, \ldots, v_n$ have been assigned the expressions $e_1, \ldots, e_n$; PC is the path condition expressing the conditions under which that path is taken; and IP refers to the next instruction to execute. Initially, the input parameters are initialized using symbols and the other variables to the special value $undef$. Symbolic execution of a program takes a symbolic state and a rule which corresponds to the

current statement referred by IP and returns the symbolic states resulting from the execution of the statement.

In order to ensure the analyzability of real-time software, the flow analysis method imposes some limitations on the handled programs. First, potentially non-deterministic and complex programming language features like recursion, dynamic memory allocation and unstructured code are not allowed. Moreover, for instance the loop induction variable update statement is limited to the form $i = a * i + b$. We think that this restriction is compatible with hard real-time systems, knowing that many works use more restrictive formulas [6]. Furthermore, the expressions can be easily extended to Presburger formulas.

4.1. Path condition and path action

Each elementary edge e in the CFG is associated a path condition $PC(e)$ which is a Boolean predicate conditioning the execution of that edge with respect to the program state s at the source node of the edge. Likewise, each basic block bb applies a block action $BA(bb)$ which represents the effect of the execution of all statements of the block on the program state s (symbolic execution rules). The path action of a block-path p denoted $PA(p)$ is the sequence of the block action of all blocks constituting that path. Likewise the path condition of a block-path is the *"logical and"* of the path condition of all edges forming that path. $PC(p) = PC(e_0) \wedge PC(e_1) \ldots \wedge PC(e_{n-1})$.

In order to compute the path action of a block-path p, we consider the set V_p of program variables assigned in different blocks of p. Let $BA(bb, v)$ be the function applied by the basic block bb on the variable $v \in V_p$ which represents the effect of the execution of all statements of the block on v. The action applied on v by p ($PA(p, v)$) is the sequence of block action applied by all blocks forming p in the order they appear in p. $PA(p, v) = (BA(bb_0, v); \ldots; BA(bb_{n-1}, v))$ is represented by an expression of the form $av + b$ (for loop induction variables) such that a and b are integer constants ($a > 0$).

4.2. Algebraic evaluation of paths

Paths are evaluated by decomposing each conditional expression $PC(p)$ into elementary Boolean expressions related by logical operators. Each conditional expression e is of the form $i\ op\ expr$, where op is a relational operator and $expr$ is an integer valued expression. For each expression e, the following parameters are evaluated:

- *Interval type:* the interval is qualified as raised if op is "$<$" or "$\leq$", constant if op is "$=$" and undervalued if op is "$>$" or "$\geq$".
- *Direction:* if the variable i is increased in the path action of p ($PA(p)$), the direction is positive and negative if i is decreased in $PA(p)$. If i is never updated along with the path, the direction is null.

The *direction* and the *interval type* are used to check for empty end unbounded paths before evaluating the number of iterations of the path. This step allows to determine the path parameters (a, b, N_1 and N_2) used by the formula 2.

In figure 3, for the state ($V = \{< i, \alpha_0 >, < n, \alpha_1 > , < cond, \alpha_2 >\}, PC = \alpha_0 \leq \alpha_1 - 1 \wedge \alpha_2 = 0$), and the path p_2^2, there are two expressions $e_1 = \alpha_0 \leq \alpha_1 - 1$ and $e_2 = \alpha_2 = 0$. The set of variables involved in e_1 is $\{i\}$, $PA(p_2^2, i) = i \leftarrow 2i$. Therefore the interval type of e_1 is *"raised"* and the direction is *"positive"*. The number of iterations is never empty nor unbounded. Then the number of iterations I_p^e of the path p related to e and the resulting symbolic state S_p^e are calculated.

In order to analytically compute the number of iterations of a loop path p, we define the following suite v_n:

$$\begin{cases} v_0 & = & N_1 \\ v_{n+1} & = & av_n + b \quad \forall n \in \mathcal{N}. \end{cases}$$

The number of iterations I is defined by the formula $N_2 - N_1 \geq \sum_{n=0}^{I-2} s_n$. $s_n = v_{n+1} - v_n$. The right-hand side of the inequality would be the greatest integer less than or equal the expression $N_2 - N_1$.

$$I = \begin{cases} \lfloor \log_a(1 + \frac{(N_2 - N_1)(a-1)}{b + (a-1)N_1}) \rfloor + 1 & when \quad N_2 > N_1 \\ \wedge\ a > 1 \wedge N_1(1-a) \neq b & \\ \lfloor \frac{N_2 - N_1}{b} \rfloor + 1 & when \quad a = 1 \\ \infty & when \quad b = N_1(1-a) \end{cases} \tag{2}$$

The number of iterations is then given by equation 2. When $b = N_1(1 - a)$, the induction variable v would have the same value during the different iterations $v_n = N_1 \ \forall n \geq 0$. Therefore, the number of iterations is unbounded ($I = \infty$). Let us consider the loop: $for(i = 1; i <= 100; i = 2 * i + 1)$, the parameters characterizing the loop path are: $a = 2, b = 1, N_1 = 1$ and $N_2 = 100$. The algebraic tool evaluates the number of iterations to 6.

The final number of iterations of the path p is determined from the number of iterations of all elementary expressions of $PC(p)$ as follows: $I_p^{e_1 \vee e_2} = \max(I_p^{e_1}, I_p^{e_2})$ and $I_p^{e_1 \wedge e_2} = \min(I_p^{e_1}, I_p^{e_2})$.

4.3. Block-based symbolic execution

The block-based symbolic execution proceeds in a post-order manner. The blocks of the level $l + 1$ are evaluated before the blocks of the level l. The evaluation of a block b is performed in the following steps:

Block	Path	Type	Path composition	Path condition	Path action	Edge
b_0^2	p_0^3	$exit$	$(e_1^3) = (BB_2)$	$cond \neq 0$		e_2^2
	p_1^3	$exit$	$(e_0^3, exit) = (BB_2, BB_3)$	$cond = 0$		$exit$
b_1^2	p_2^3	$exit$	$(e_3^3) = (BB_4)$	$i \geq n$		e_4^2
	p_3^3	$exit$	$(e_2^3, e_4^3) = (BB_4, BB_5)$	$i \leq n-1$	$(i \leftarrow 2i)$	e_3^2
b_0^1	p_0^2	$exit$	$(e_1^2) = (BB_1)$	$i \geq 2n$		$exit$
	p_1^2	$exit$	$(e_0^2, exit) = (BB_1, b_0^2)$	$i \leq 2n-1 \wedge cond \neq 0$		$exit$
	p_2^2	$loop$	$(e_0^2, e_2^2, e_3^2) = (BB_1, b_0^2, b_1^2)$	$i \leq n-1 \wedge cond = 0$	$(i \leftarrow 2i)$	
	p_3^2	$loop$	$(e_0^2, e_2^2, e_4^2, e_5^2) = (BB_1, b_0^2, b_1^2, BB_6)$	$n \leq i \leq 2n-1 \wedge cond = 0$	$(i \leftarrow i+1)$	
b_0^0	p_0^1	$exit$	$(start, e_0^1, exit) = (BB_0, b_0^1)$	$true$	$(i \leftarrow 1; PA(b_0^1))$	e_1^0
top	p_0^0	$exit$	$(e_0^0, e_1^0) = (start, b_0^0, exit)$	$true$	$(PA(b_0^0))$	

Table 1. Path definition and parameters.

a- Path information The first step consists of determining a set of information characterizing the block. We determine the set of *block-paths* of the block. For each path a set of parameters is calculated: path type (*loop* or *exit path*); the set of edges forming the path; path condition; path action; and finally for exit paths, the edge of the enclosing block graph on which the flow will go after taking that path. The starting point of that edge constitutes an *exit point* of the block. This information is kept in a table where a summary for the example of the figure 1 is shown in table 1.

b- Block evaluation The symbolic execution of a block b is performed by evaluating all the *block_paths* $P = block\_paths(b)$ starting at the entry point of the block with a symbolic state in which all variables used in the block are assigned symbols. For evaluating each path p we use a symbolic state in which PC corresponds to the path condition of p. This step yields the set of the block exit states S_b and the number of iterations of the block (figure 3).

c- Path evaluation The path evaluation takes a symbolic state s, the path action $PA(p)$ and performs the algebraic evaluation of the path using the formula 2. The result is the number of iterations of the path I_p and the generated symbolic states S_p.

Figure 3 illustrates the block-based symbolic execution of the block b_0^1. Edges are annotated by the number of iterations applied by the path on the symbolic state of the starting node. We assume that the variable $cond$ is updated in the *"instructions"* block but not n. The resulting symbolic states are the terminal nodes ($s_0^t, s_1^t, \ldots$) (figure 3). Merging of states may be performed which allows to reduce the number of resulting states. Two states $s_1 =< V_1, PC_1, IP >$ and $s_2 =< V_2, PC_2, IP >$ with the same instruction pointer IP can be merged into one state $s =< V_1 \cup V_2, PC_1 \vee PC_2, IP >$. Merging of states may sometimes cause information loss. Then some pessimism will be incurred in the WCET estimate. Therefore, a trade-off must be done between the WCET precision and the number of generated states. States s_1^t and s_2^t may be merged into one state s_3^t ($< V = \{< i, \alpha_0^1 >, < n, \alpha_1 >< cond, \alpha_2^* >\}, PC = \alpha_0 \leq \alpha_1 - 1 \wedge \alpha_1 \leq \alpha_0^1 \leq 2\alpha_1 >$).

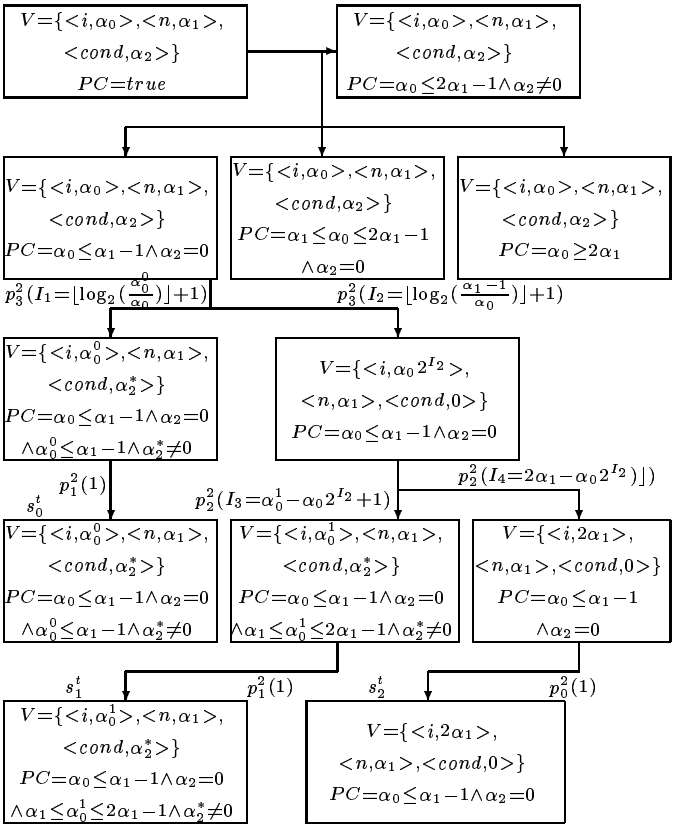


Figure 3. Block-based symbolic execution of the block b_0^1

These states constitute the block exiting symbolic states which would be used in the evaluation of lower level blocks (ex. b_0^0). Indeed, when the current block b^l is examined in the framework of a block b^r of lower level, the block action of the block $BA(b^l)$ is evaluated in one step.

When the execution reaches the block b_0^0 (higher level), the variable i is initialized to 1, then the incomplete branches of the figure 3 are discarded (PC becomes $false$), reducing thus the number of the block exiting states to only 3. Furthermore, it is possible to keep only the resulting states maximizing the WCET of the block.

4.4. Iterative blocks with variant number of iterations

In the case of nested loops, the number of iterations of an inner loop may depend on the control variables of outer loops and thus varies following those dependencies. The worst-case number of iterations for such a block may be considered always its limit. This may result in an important WCET over-estimation. Therefore, the number of iterations of an inner loop must be expressed in terms of control variables of outer loops values. The block-based symbolic execution approach is able to estimate a worst-case number of iterations of such blocks without over-estimation.

Assume that the *"statements"* area of the figure 1-a comprises a block b_0^3 consisting of the loop: $for(j = 0; j < i; j++)$. One can estimate the WCET of the loop nest to $2n \times (2n-1)$ since i starts with the value 1. When applying the block-based symbolic execution: $s_{in} =< \{< i, \alpha_0 >, < n, \alpha_1 >, < j, \alpha_3 >\}, PC = true >$, the results are $s_{out} =< \{< i, \alpha_0 >, < n, \alpha_1 >, < j, \alpha_1 >\}, PC = true >$ with a number of iterations of $\alpha_1 - \alpha_0$. When the analysis reaches the top level (i is initialized to 1), the number of iterations is evaluated to $\sum_{i=1}^{2^{I_2}} i[i \rightarrow 2i] + \sum_{i=2^{I_2}}^{2n-1} i$. For example, for $n = 10$, the intuitive method yields 380 while our method provides the actual WCET 85.

In addition, the block based symbolic execution eliminates implicitly most of the program infeasible paths and allows to express the WCET estimates as symbolic expressions function of the program parts input parameters (function parameters, etc.). The quality of the provided flow information is comparable to the one of symbolic execution and abstract interpretation schema since our approach is a symbolic execution method.

5. Conclusion

WCET analysis is a popular method used to validate the temporal correctness of real-time systems. WCET analysis may be done statically on the program source or object code, which results in overestimated values. Therefore, techniques allowing to tighten the WCET estimates are required. However, these techniques are complex because they deal with program semantics.

We proposed a practical approach aimed to automatically extract flow information related to program semantics which will be used to tighten the WCET estimates. The method presents a reduced complexity in terms of time and memory by avoiding unfolding iterative blocks. Moreover, the approach provides tight values since it handles non rectangular loops and loops with multiple exit conditions and eliminates implicitly most of the infeasible paths.

We are implementing a prototype of the method in order to evaluate its performance. Furthermore, we plan to extend the expression used to evaluate loops to Presburger formulas and use the results obtained on those formulas.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[2] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

[3] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *8th European software engineering conference*, pages 142–151, New York, USA, 2001. ACM Press.

[4] A. Ermedahl. *A modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[5] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2):61–74, 1998.

[6] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, , and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2-3):129–156, May 2000.

[7] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-time Systems, La Jolla, California*, June 1995.

[8] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003*, pages 99–102, Polytechnic Institute of Porto, Portugal, 2003.

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

[10] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Journal of Real-Time Systems*, 1(2):160–176, September 1989.

[11] P. Puschner and A. V. Schedl. Computing maximum task execution- a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

A Framework for the Busy Time Calculation of Multiple Correlated Events

Simon Schliecker, Matthias Ivers, Jan Staschulat, Rolf Ernst
Institute for Computer and Communications Network Engineering
{schliecker, ivers, staschulat, ernst}@ida.ing.tu-bs.de

Abstract

Many approaches to determine the response time of a task have difficulty to model tasks with multiple memory or coprocessor accesses with variable access times during the execution. As the request times highly depend on system setup and state, they can not be trivially bounded. If they are bounded by a constant value, large discrepancies between average and worst case make the focus on single worst cases vulnerable to overestimation.

We present a novel approach to include remote busy time in the execution time analysis of tasks. We determine the time for multiple requests by a task efficiently and and far less conservative than previous approaches. These requests may be disturbed by other events in the system. We show how to integrate such a multiple event busy time analysis to take into account behavior of tasks that voluntarily suspend themselves and require multiple data from remote parts of the system.

1 Introduction and Overview

The analysis of the worst case timing behavior of systems is facing new challenges with increasing system complexity. In order to derive reliable bounds, overestimations must often be introduced to reduce the analysis complexity. However, this will either increase costs or thwart industrial use altogether. Therefore, timing analysis must be sure to cover realistic system setups with tight timing bounds.

A particular challenge is the behavior of tasks that strongly interact with their environment during execution, e.g. through memory or coprocessor requests: The waiting for such external requests introduces additional delays. Thus the execution time can not be known without knowledge of these delays, which depend highly on system setup and state. Furthermore, if the scheduler reallocates the processor to other tasks, conserving the processor time, but also additionally delaying the requesting task, the response time can not be determined on the basis of the tasks core execution time alone.

Also, the focus on absolute worst case response times in system level analysis has impaired the analysis potential of such tasks: Assuming requests to cause a constant delay to the execution leads to a large overestimation in shared resource multi-task environments, where the worst case can outgrow the common case by very large factors.

The contribution of this paper is a new method to investigate communicating tasks which issue a large number of events during execution. We present methods to derive the total busy time of an execution seperated into multiple chunks, as well as the total busy time of multiple transactions over multiple resources. Both is integreated to find response times of communicating tasks. We closely examine a static priority preemptive (SPP) scheduler and show the improvement over previous work in experiments.

This paper is organized as follows: We will present related work on timing analysis in Section 2 and a new model for communicating tasks in Section 3. Section 4 presents our framework, which is implemented for a SPP scheduler in Section 5. We present an example and experiment in Section 6, and conclude in Section 7.

2 Related Work on Timing Analysis

Timing of real-time systems is addressed on different levels of abstraction. We will first present approaches that closely examine the *tasks* internal behavior. Approaches that work on the *resource* level take these results as the basis for a schedulability analysis. Finally, *system* level approaches investigate the system behavior to derive timing properties such as path latencies.

The timing analysis of *individual tasks* is commonly separated in two stages [7] [6]: Microarchitecutural modeling, in which the timing of sequences of instructions is investigated, and program path analysis to determine which path is executed in the worst-case. Memory or coprocessor access times, or cache miss penalties are assumed to be constant parameters in most approaches.

The interference of *multiple tasks on the same resource* is considered in the response time analysis. The growing-window technique is the prevailing method to solve worst case response time equations which do not lend themselves easily to direct solution. Originally introduced in [5], it has been extended to include arbitrary arrival patterns and additional timing effects e.g. in [8].

In a simple version, with no blocking time and no multiple releases within a busy window the worst case response

time $WCRT(\tau)$ of a task τ with worst case execution time $C(\tau)$ on a resource with SPP scheduling is given by the smallest time w that fulfills the following equation:

$$w = C(\tau) + I_{hp}(w) \quad (1)$$

where $I_{hp}(w)$ is the worst case interference τ can experience due to the execution of higher priority tasks within a time window of size w.

Bletsas et al. have shown in [1] how to consider tasks whose execution is separated into actual execution times and known communication times in the response time analysis. Their approach accounts for the parallelism in local and remote execution, in so far that the interference by higher priority tasks is reduced by the "gaps" during which they wait for remote data. Still, the gap time is assumed to be constant and independant of previous behavior, which is not the case, e.g. when requests by different tasks are pipelined.

System level analysis is necessary to derive the *path latency* of memory transactions, or other requests that pass over multiple components of a system. To avoid confusion with paths within a tasks control flow, we call paths through the system *chains*.

The classical worst case response time calculation was extended to distributed systems in [9]. Other approaches, such as [4] break down the analysis complexity of complete systems into separate local analyses and bind them together with a description of the traffic (*event streams*). Attributing value dependent execution times (*modes*) to tasks [3] can lead to better local response times, if the event streams are enriched with a description of the type of events in the stream. Any of these conservative approaches focus solely on the worst case time of any single event.

The strict distinction between the different levels of abstraction was broken down in [2], where an integrated approach to perform program path analysis and derive coprocessor request latencies was presented, by assuming a worst case scenarios for each request.

3 Communicating Tasks

The presence of communicating tasks, which perform system-wide requests during their execution, contradicts a number of the assumptions of the classical analysis distribution, in which only "bottom-up" dependencies exist. The main problem is that the worst case execution behavior depends on the system level influences that can not easily be bounded before the execution behavior is known. If a conservative worst case can be found at all, it is many cases a high overestimation of the average case. Also, voluntary suspension of tasks can lead to additionaly scheduling delays. We will therfore introduce a model for communicating tasks, that enables improved reasoning about the distributed execution behavior.

The traditional task concept assumes tasks consisiting of basic blocks of linear code, branches, and loops, all represented in a control flow graph. Some worst case input pattern leads to a worst case timing behavior as shown in Figure 1a. The timing includes execution of instructions on the processor as well as memory or coprocessor calls.

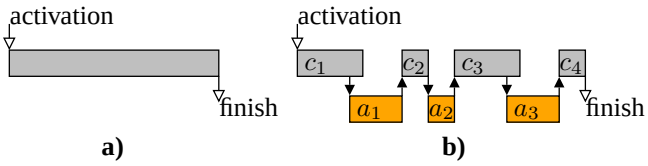


Figure 1. Task execution model.

We assume that a *communicating task* performs data requests by initiating *transactions* (a_1 to a_3) though executing an explicit instruction (`CALL a`), where `a` defines the target and type of the transaction. By calling an instruction `SYNC a`, the task will be suspended until completion of the transaction. We call the parts of a task during which no external data is required *consecutive execution sequences* (CES c_1 to c_4), each of which can be seen as corresponding to the classical task concept. A CES will often be computation but may also be communication or data storage, depending on the type of resource the CES is executed on.

A worst case task behavior exists that maximizes the time until completion, including finishing of all CESs and transactions. For the scope of this paper we assume that the maximum execution time of each CES is known, and to reduce the complexity of our problem that the amount and type of incurred transactions is not data dependant. This behavior is sketched in Figure 1b.

A transaction consists of an ordered set of *events*. Each event is the signal that causes one CES on any resource to become *ready*. This CES is then *processing* the event until it is *finished* and thus not ready anymore. When the CES is finished, the next event of the transaction becomes *ready*, activating the next CES. The first event of the transaction *initiates* the transaction and is given by the `CALL` instruction. The transaction is *finished* when the last event of the transaction was processed. When a transaction is initiated but not finished it is *ongoing* or *ready*.

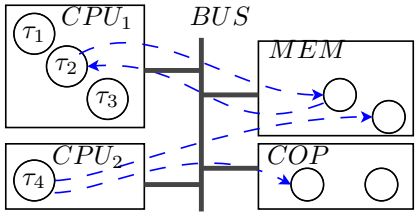


Figure 2. A Multiprocessor Setup.

Figure 2 shows the setup of an example multiprocessor system. Three tasks are mapped to processor CPU_1. Dur-

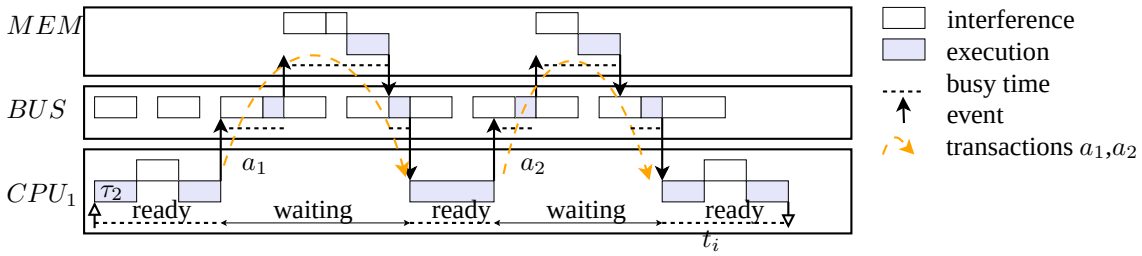


Figure 3. A distributed execution

ing its execution, τ_2 requires data from the memory MEM. A task τ_4 on CPU_2 also uses the bus and memory, interfering with the communication of τ_2. Figure 3 shows a possible Gantt diagram of this example. Task τ_2 on CPU_1 requests data from the memory two times, each time initiating a *transaction* consisting of 4 *events*. The processing of the events is delayed on the BUS and on the memory MEM, due to τ_4 performing similar accesses. As the overall time window is larger than without transactions, increased interference (for example by higher priority tasks as experienced by τ_2 at time t_i in Figure 3) occurs. The dotted lines denote the actual *overall busy time* of τ_2 which is the focus this paper.

4 A Framework for Worst Case Busy Times

We present a coupled analysis, that integrates the tasks execution behavior with the system level behavior to find the tasks response time. We set a specific focus on the analysis of multiple transaction on the system level, and the seperation of tasks into multiple parts on the local level, which is the common scenario for tasks with remote data requirements.

First, a worst case busy time analysis is introduced in Section 4.1, which allows tasks to request data multiple times during execution. This approach relies on the calculation of a local total busy time (generally addressed in Section 4.2, and specifically for SPP in Section 5), and the busy time for the memory transactions, which is calculated in Section 4.3.

4.1 Worst Case Busy Time Analysis

Extending the scope of the response time analysis from single worst case behavior to conservative bounds of multiple events requires the introduction of new terminology. Let the set of CESs of a task that leads to the largest local execution time be denoted by $\mathbb{E}$ and the set of transactions that is initiated by a task denoted by $\mathbb{Q}$. Furthermore:

The *total busy time of a set of CESs* $\mathbb{E}$ is the total amount of time during which at least one CES of $\mathbb{E}$ is ready. Thus this is the union of the times during which any single CEC is ready.

The *total busy time of transactions* $\mathbb{A}$ is the total amount of time during which at least one transaction of $\mathbb{A}$ is ready (i.e. ongoing).

The *overall busy time* is the total amount of time during which either a CES or a transaction is ready.

Figure 3 shows the execution of task τ_2, which consists of 3 local CESs on CPU_1 and initiated transactions a_1 and a_2. The dotted lines comprise the overall total busy time of the tasks CECs and transactions. The following theorem gives a overall busy time based on the definitions above.

Theorem 1. *The overall busy time of a task τ executing on resource r during a time window of size $w \geq 0$ is given by w such that*

$$w = S^r_{CES}(\mathbb{E}, w) + S_{trans}(\mathbb{A}, w) \tag{2}$$

$\mathbb{E}$ *is the set of CESs that the task τ needs to execute locally for completion and*

$\mathbb{A}$ *is the set of transactions the task τ initiates and requires to complete execution.*

$S^r_{CES}(\mathbb{E}, w)$ *is the maximum total busy time for CESs $\mathbb{E}$ and*

$S_{trans}(\mathbb{A}, w)$ *is the maximum total busy time for transactions $\mathbb{A}$ under the assumption that all CESs and transactions can be finished within time w.*

Proof. Assume that transactions are initiated at the very last instant of each CES of τ. From the definition it follows that as soon as the transaction is finished the next CES is ready to be executed. This means that whenever no CES is ready, a transaction must be ongoing or the task is finished.

As the maximum total amount of time that transactions can be ongoing is bounded by $S_{trans}(\mathbb{A}, w)$ and the maximum amount of time CESs are ready is bounded by $S^r_{CES}(\mathbb{E}, w)$, it follows that after $S^r_{CES}(\mathbb{E}, w) + S_{trans}(\mathbb{A}, w)$ the task must be finished.

If a task issues the transactions not at the end of a CES but earlier, both the task and a transaction are ready at the same time. This can only lead to a smaller total busy time. □

To solve equation 2 a growing-window technique as in [8] can be used. Initially, a non-conservative value for w can be picked. It will not be possible to perform all requested computation in time, as it will take at least until $S_{CES} + S_{trans}$ to finish. This value is used as a new w and tested. As soon as w is large enough to contain all busy times, the assumptions are correct and the analysis has converged. An example of this procedure is given in Section 6.

The above theorem is valid independently of the utilized arbitration policies, as S_{CES} and S_{trans} are unspecific. The next sections 4.2 and 4.3 focus on the derivation of these values.

4.2 Multiple Event Busy Times

One property of the memory and coprocessor requests adressed in this paper is that commonly many occur during a tasks execution. We will investigate the case of the execution of multiple CESs in a given time window. Figure 4 shows the busy time of 4 CECs that are processed in a first-in-first-out ordering. The resource is also handling requests by other tasks, which leads to delay due to interference. The dotted line depicts the searched total busy time, which is the union of the individual CESs' busy intervals (R_1 to R_4).

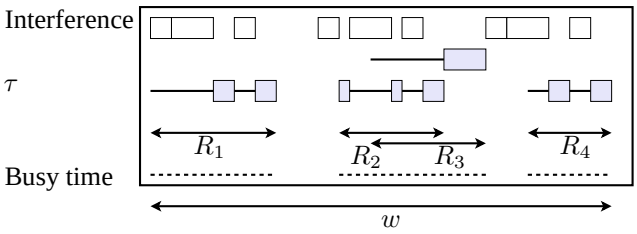


Figure 4. Single Resource Total Busy Time.

The total busy time can be used for two things: Firstly, it gives the maximum total amount of time during which this resource can be busy processing CECs that are activated by events that are part of a transaction. This is required to determine the remote total busy time S_{trans} of Theorem 1 and is investigated in Section 4.3.

Secondly, it is also an estimate on the local total busy time S_{CES} of a communicating task. The execution of such a task consists of CESs $\mathbb{E}$ (see Section 3). Although the CESs of a single task may not overlap, the total busy time is still valid.

Tindells approach to response times for bursty job arrivals [8] is not applicable, as it finds the worst case response time only within a self-inflicted and continuous *busy window*. In our case however nothing is said about the arrival times, so that events may *also* arrive completely *separated*.

Anyway, the worst case response time as derived in [8] and similar approaches may be reused: For any scheduling arbitration the maximum total busy time is bounded by the sum of the individual worst case response times $WCRT(c)$ of the CESs $c \in \mathbb{E}$.

$$S^r_{CES}(\mathbb{E}, w) \quad = \quad \sum_{c \in \mathbb{E}} WCRT(c) \tag{3}$$

Similarly, the busy time required to process a set of events $\mathbb{Q}$ to CESs on the same resource can be bounded. As each event $q \in \mathbb{Q}$ causes one CES, $c(q)$, to become ready, the set of CESs to process is given by

$$\mathbb{E} = \bigcup_{q \in \mathbb{Q}} c(q) \tag{4}$$

Note that Eq. 3 is an overestimation as can be seen in Figure 4. Firstly, not every event must wait for the processing of previous events to be finished as is assumed in the calculation of $WCRT(c)$, rather the individual busy windows overlap. Furthermore, not every request can experience the critical instant of interference by other tasks, but only a certain amount of interference can occur in in the given time window. We will therefore present an improved analysis specifically for static priority preemptive (SPP) scheduling in Section 5.

4.3 Busy Time of Transactions

In the previous section we have investigated processing of multiple events on the same resource. We will now turn to the total busy time of transactions that consists of multiple events on different resources, S_{trans}, in Eq. 2. For this, we can build on the results of section Section 4.2 (and 5).

A straightforward bound for the total busy time of transactions $\mathbb{A}$ is the sum over the worst case response times that each individual transaction would have taken. This is again an overestimation, as the worst case interference can in many cases not be imposed on every single transaction, and not every transaction may be delayed by preceding transactions. An amelioration is achieved by Theorem 2, where multiple transactions are investigated together.

Theorem 2. *Let each transaction in $\mathbb{A}$ consist of events to CESs mapped to a set of resources $\mathbb{R}$, and it is known they can be initiated and finished within a time window of size w. Let all events of the transaction on the same resource be treated with a first-in-first-out principle. Then the maximum total busy time of the transactions $\mathbb{A}$ is given by:*

$$S_{trans}(\mathbb{A}, w) \quad \leq \quad \sum_{r \in \mathbb{R}} S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w) \tag{5}$$

where $\mathbb{E}^{\mathbb{A}}_r$ is the union of all CESs that are executed on r and activated by an event of the transaction $\mathbb{A}$ and $S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w)$ is the maximum total busy time of these CESs.

Proof. Let $\mathbb{T}_r(c)$ be the time interval at which a specific CES c on resource r is ready. The total amount of time that resource r can be busy processing events is given by the size of the union of all times at which at least one CES of the transaction is ready on r.

As the transactions are assumed to be finished within a time window of size w, Theorem 3 bounds the total busy time of the CESs which correspond to the events in the transaction by $S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w)$.

$$\left| \bigcup_{c \in \mathbb{E}^{\mathbb{A}}_r} \mathbb{T}_r(c) \right| \quad \leq \quad S^r_{CES}(\mathbb{E}^{\mathbb{A}}_r, w) \tag{6}$$

$\bigcup \cdot$ Produces the union of included intervals.
$|\cdot|$ Is the sum of the total size of all included intervals.

A transaction along the chain is ongoing whenever an event of the transaction is ready on any of the given resources along the chain. Therefore, the busy time of the transactions $\mathbb{A}$ is bounded by

$$S_{trans}(\mathbb{A}, w) = \left| \bigcup_{r \in \mathbb{R}} \bigcup_{c \in \mathbb{E}_r^{\mathbb{A}}} \mathbb{T}_r(c) \right| \quad (7)$$

As the size of the union of intervals can not be larger than the sum over the sizes of the intervals, equation 5 follows from equations 6 and 7. □

This framework allows to determine the worst case response time of tasks which require multiple data from other parts of the system, and initiate transactions to fetch this data. The total busy time of the transactions was determined, and the effect of the resulting voluntary suspensions of the task into multiple CESs has been taken into account in Section 4.2, albeit rather imprecisely. We will now improve the considerations about local total busy time in the following section.

5 A Static Priority Preemptive Scheduler

To show the validity of the approach presented in Section 4, we introduce a simple static priority preemptive (SPP) scheduler that arbitrates tasks with transaction and resynchronization instructions. Based on SPP scheduling, the scheduler ensures that at every time point the task with the highest priority that has all data required for execution, and has not completed execution is executing on the resource. Tasks may consist of multiple CECs, that receive the same priority as the task. All CECs with the same priority are treated first-in-first-out. For the scope of this paper, we assume that there is no blocking caused by shared critical sections, and the scheduling procedure induces no significant additional overhead.

Theorem 3. *Let a set of CESs $\mathbb{E}$ have the same priority on resource r that is scheduled with mechanisms described above. Let the processing of all CESs be started and finished within a time window of size w. Furthermore, let $C(c)$ be the worst case computation time of a CES $c \in \mathbb{E}$, and $I_{hp}(w)$ be the maximum time tasks with higher priority may be executing. Then the maximum total busy time of $\mathbb{E}$ is given by the following equation:*

$$S_{CES}^r(\mathbb{E}, w) \quad = \quad \sum_{c \in \mathbb{E}} C(c) \quad + I_{hp}(w) \quad (8)$$

Proof. The total busy time B is given by the sum of all times during which at least one CES is ready. Let $\text{RUN}(t)$ be the task or CES chosen by the scheduler to execute at time point t.

All times t at which $\text{RUN}(t) \in \mathbb{E}$, a CES in $\mathbb{E}$ is being executed and must therefore be ready, thus t must be included in B. This can be the case for at most $\sum_{c \in \mathbb{E}} C(c)$.

At times t, when $\text{RUN}(t) \notin \mathbb{E}$, either no CES in $\mathbb{E}$ is ready, or if one ore more CES is ready, they are kept from executing by a higher priority task that is ready. Whenever no CES in $\mathbb{E}$ is ready, B does not increase. The total amount of time higher priority tasks can be executing is limited by $I_{hp}(w)$, and in the given scheduler, at least one higher priority task is ready, only if a higher priority task is executing. Thus, whenever a CES in $\mathbb{E}$ is ready, it can not be kept from executing for more than $I_{hp}(w)$ within a time interval of size w.

Thus, the CESs in $\mathbb{E}$ can not be ready for more than $\sum_{c \in \mathbb{E}} C(c) \quad + I_{hp}(w)$ in a time interval of size w. □

Compared to section 4.2, this is a better estimate of the busy time of the CECs in $\mathbb{E}$, as now the worst case interference in the given time window w in which all processing takes place is only accounted once.

Note that the worst case interference $I_{hp}(w)$ by higher priority tasks which are allowed to suspend themselves to request data is not given by the traditional "critical instant" of all tasks being activated simultaneously [1]. Instead, the first interfering invocation has to be assumed to have performed all suspension *before* the beginning of the time window, which leads to an earlier possible activation of all successive invocations.

6 Example and Experiments

Consider the System in Figure 2 and 3. Let all resources be scheduled with the SPP scheduling as described in Section 5. We are interested in the response time of τ_2. Assume that τ_2 initiate 5 transactions $\mathbb{A}$ to the memory. Let the period and jitter be according to Table 1, and the deadline of τ_2 equal to its period. Assume that τ_1 is the only task on CPU_1 with a higher priority. On both the bus and the memory, the priority of the tasks handling τ_2's transactions are lower than the interference (I_4^1, I_4^2, and I_4^3) caused by the transactions of τ_4 on CPU_2, which occur with the period and jitter as shown. Besides their transactions, let τ_1 consist of a single CES of size 10, and τ_2 of one of size 50. Let the execution time of any execution on the bus or the memory be 10.

	CPU1		**Bus**			**Memory**	
	τ_2	τ_1	$\mathbb{A}$	I_4^1	I_4^2	$\mathbb{A}$	I_4^3
Period	400	100	n/a	100	100	n/a	100
Jitter	0	200	n/a	0	200	n/a	0

Table 1. Example Setup

A traditional response time analysis utilizes the worst case time for a single request, each delayed by the maximum amount of interference. This calculates to $WCRT_{BUS} + WCRT_{MEM} + WCRT_{BUS} = 50 + 40 + 50$ (calculation not shown). Thus for 5 requests a time of $5 * 140 = 700$ is required. Based on this, τ_2 can not keep its deadline.

Determining the worst case response time on the basis of multiple event busy times yields much tighter bounds. According to Theorem 1, the overall busy time is given by $S_{CES}^{CPU_1} + S_{trans}$, where $S_{CES}^{CPU_1}$ is denoted by S_{CPU} and S_{trans} according to Theorem 2 is given by $S_{BUS} + S_{MEM}$. Initially, it is assumed that all computation request can be handled within a time window size of $w = 50$, which is the core time of τ_2. If this were the case, the computation on CPU_1, Bus, and the memory would take 80, 130 and 80 time units respectively. Thus, the computation can not be finished within the assumed time window. A new test is done with the time window size 290, which also fails. This goes on until finally it is assumed that all computation is started and finished in a time window of size 380, thus the assumption holds, and a the worst case response time of τ_2 is found. This is an improvement towards previous approaches, as the interference in the overall busy window is only accounted for once.

	CPU1		**Bus**			**Memory**	
	$C(\tau_2) = 50$		$C(q) = 10$			$C(q) = 10$	
w	I_1	S_{CPU}	I_1	I_2	S_{BUS}	I_4	S_{MEM}
50	30	80	15	15	130	30	80
290	50	100	25	25	150	50	100
350	60	110	30	30	160	60	110
380!	60	110	30	30	160	60	110

Table 2. Calculation Procedure

We conducted a set of multiple request experiments to show the gain of our analysis ins Section 5 over the sum of worst cases approach. Figure 5 shows the estimated response times for a number of requests, from which it is known that they occur within a time window of size 300. As the number of requests increases, the total busy time only increases by the added core execution time. This results from the fact, that the complete possible interference in the time window is assumed from the beginning. This is also the reason, why the new method is inferior for $N = 1$, as in traditional WCRT only the interference during the requests busy window (not w) can interfere with the execution. As both approaches are conservative, the minimum can be used for an optimal analysis. The positive effect scales for transactions as suggested by the aboeve example.

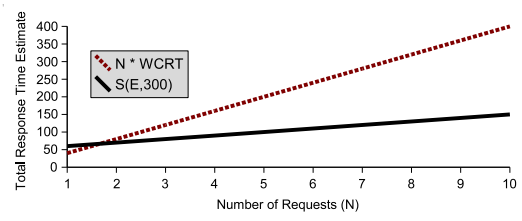


Figure 5. Multiple Request Total Busy Time.

7 Conclusion

To allow the analysis of communicating tasks that consist of local execution sequences and remote transactions we have proposed a framework that integrates the analysis over different levels of abstraction. We address multiple events together and calculate an upper bound on the total amount of busy time. This is both used for a tight estimate on the transaction latencies as well as the overall time to complete the task execution. Additionally, we presented a straight-forward analysis that accounts for the properties of static priority preemptive scheduling. For scheduling policies, where no such adopted analysis is possible or available, we have presented a conservative fall-back solution.

The experiments have shown how the consideration of a larger time window and multiple events can significantly improve the estimates on the worst case response time of a task that issues multiple memory requests.

References

[1] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, Catania, Italy, jul 2004. IEEE Computer Society, IEEE.

[2] M. Ivers J. Staschulat, S. Schliecker and R. Ernst. Analysis of memory latencies in multi-processor systems. In *WCET Workshop*, Palma de Mallorca, Spain, July 2005.

[3] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proceeding Design Automation and Test in Europe*, Paris, France, March 2004.

[4] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[5] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29(5):390–395, October 1986.

[6] Y.-T. S. Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, 1996.

[7] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by seperate cache and path analyses. *Real-Time Systems*, 18(2/3), May 2000.

[8] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[9] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3):117–134, apr 1994.

Towards Formally Verifiable WCET Analysis for a Functional Programming Language

Kevin Hammond* Christian Ferdinand † Reinhold Heckmann† Roy Dyckhoff*
Martin Hofmann‡ Steffen Jost* Hans-Wolfgang Loidl‡ Greg Michaelson§
Robert Pointon§ Norman Scaife¶ Jocelyn Sérot¶ Andy Wallace§

Abstract

This paper describes ongoing work aimed at the construction of formal cost models and analyses to yield verifiable guarantees of resource usage in the context of real-time embedded systems. Our work is conducted in terms of the domain-specific language *Hume*, a language that combines *functional programming* for computations with *finite-state automata* for specifying reactive systems. We outline an approach in which high-level information derived from source-code analysis can be combined with worst-case execution time information obtained from high quality abstract interpretation of low-level binary code.

1 Introduction

The EU Framework VI EmBounded project (IST-2004-510255) aims to automatically determine strong resource bounds for high-level programming language features. We aim to obtain formally verifiable certificates of bounds on resource usage from a source program through automatic analysis.

1.1 The Hume Language

Our work is undertaken in the context of Hume [13], a functionally-based domain-specific high-level programming language for real-time embedded systems. Hume is designed as a layered language where the *coordination layer* is used to construct reactive systems using a finite-state-automata based notation; while the *expression layer* is used to structure computations using a strict purely functional rule-based notation that maps patterns to expressions. The coordination layer expresses reactive Hume programs as a static system of interconnecting *boxes*. If each box has bounded space cost internally, it follows that the system as a whole also has bounded space cost. Similarly, if each box has bounded time cost, a simple schedulability analysis can be used to determine reaction times to specific inputs, rates of reaction and other important real-time properties. Expressions can be classified according to a number of levels where lower levels lose abstraction/expressibility, but gain in terms of the properties that can be inferred. For example, the bounds on costs inferred for recursive functions will usually be less accurate than those for non-recursive programs, and cannot always be deduced.

Previous papers have considered the Hume language design in the general context of programming languages for real-time systems [13], and specifically functional notations for bounded computations [12], described a heap and stack analysis for FSM-Hume [14], and considered the relationship of Hume with classical finite-state machines [17]. The main contribution of this paper is to outline a worst-case execution time analysis for Hume combining high- and low-level information, where source-level information on the costs of recursive functions, conditional expressions etc. is combined with machine-level information on cache behaviour, pipelines etc.

We will illustrate the design of Hume with a simple control example for a reactive system: the controller for a drinks vending machine. Figure 1 shows the Hume box diagram for this system, and the code is shown below. Note that `*` in an input or output position is used to indicate that that position is ignored, and that all inputs and outputs are matched asynchronously.

```
type Cash = int 8;

data Coins =  Nickel | Dime;
data Drinks = Coffee | Tea;
data Buttons = BCoffee | BTea | BCancel;

-- vending machine control box
```

*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX.
email: {kh,rd,jost}@dcs.st-and.ac.uk.
†AbsInt GmbH, Saarbrücken, Germany.
email: {cf,heckmann}@absint.com
‡Ludwig-Maximilians Universität, München.
email: {mhofmann,hwloidl}@informatik.uni-muenchen.de
§Depts. of Comp. Sci. and Elec. Eng., Heriot-Watt University, Riccarton, Edinburgh, Scotland.
email: {G.Michaelson, A.M.Wallace}@hw.ac.uk
¶LASMEA, Université Blaise-Pascal, Clermont-Ferrand, France.
email: Jocelyn.SEROT@univ-bpclermont.fr
This work has been supported by EU Framework VI grant IST-2004-510255 and by EPSRC Grant EPC/0001346.

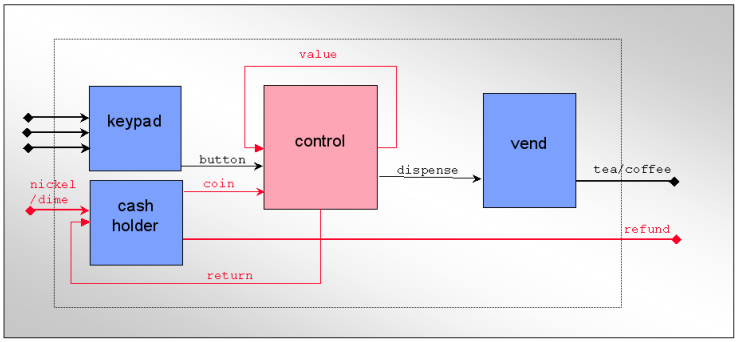


Figure 1: Hume example: vending machine box diagram

```
box control
in  ( coin :: Coins, button :: Buttons,
                           value :: Cash )
out ( drink :: Drinks, value' :: Cash,
                           return :: Cash )
match
    ( Nickel, *,  v )  ->  ( *, v +  5, * )
  | ( Dime,   *,  v )  ->  ( *, v + 10, * )
  | ( *, BCoffee, v )  ->  vend Coffee 10 v
  | ( *, BTea,    v )  ->  vend Tea 5 v
  | ( *, BCancel, v )  ->  ( *, 0, v )
  ;

vend drink cost v =
   if v >= cost then ( drink, v-cost, * )
                else ( *,     v,      * );
```

Functional languages have rarely been applied to hard real-time systems, partly because they are perceived as hard to cost. The most widely used *soft* real-time functional language is the impure, strict language Erlang [1], which has been succesfully used in many large commercial applications. However, there have also been attempts to apply pure functional languages to soft real-time settings (e.g. [24, ?, 25]). Few, if any, of these approaches provide strong cost models, however. Synchronous dataflow languages such as Lustre [6] or Signal [11] have strong similarities with a functional approach, being similarly declarative. The primary difference from our approach is that Hume also supports asynchronicity, is built around state machines, and provides a highly-expressive programming environment including rich data structures, recursion, and higher-order functions, while still providing a strong cost model.

2 A Source-Level Cost Model for Hume Expressions

Our approach involves producing a formally verifiable upper bound cost model for Hume programs that is related to actual execution costs. We can use this model to produce high-level static analyses for determining bounds on recursive calls, iterative loops etc. In this paper, we use an abstract machine approach, where execution costs are associated with abstract machine instructions, and where these costs are related to Hume source forms through formal translation. In this way, the mathematical cost model can be insulated from changes in the concrete architecture, being effectively parameterised by cost information for each abstract machine operation. A *correspondence proof* (omitted here, for brevity) formally relates the high-level model to costs expressed in terms of the abstract machine. In this way, we are able to prove that the source level timing information we give here is an upper bound on actual exection costs, provided only that the timing information for each abstract machine instruction is a true upper bound on the execution cost of that instruction, including effects of pipelining and cache behaviour. The use of purely functional expressions within Hume boxes simplifies both the construction of the cost model and the corresponding proofs, by avoiding the need for dataflow analysis, alias analysis and other consequences of the use of side-effects. This also improves the accuracy of the result.

As a proof-of-concept, we have chosen a simple, high level stack-based machine, the Hume Abstract Machine (or HAM). The approach can, however, be generalised to other abstract machine designs or to direct compilation, as required.

The formal statement $\mathcal{V}, \eta \vdash^{t\,|\,p\,|\,m}_{t'\,|\,p'\,|\,m'} e \rightsquigarrow \ell, \eta'$ may be read as follows: given the value environment $\mathcal{V}$ and initial heap η, expression e evaluates in a finite number of steps to a result value stored at location ℓ in the modified heap η', provided that there were t time, p stack and m heap units available before computation. Furthermore, at least t' time, p' stack and m' heap units are unused after evaluation. We illustrate the approach by showing a few sample rules covering key expression forms.

Variables are simply looked up from the environment and the corresponding value pushed on

the stack. The time cost of this is the cost of the `PushVar` instruction, shown here as `Tpushvar`. Concrete values for this constant can be obtained using either measurement-based approaches [2] or abstract interpretation (see Section 3). There is no heap cost.

$$\frac{\mathcal{V}(x) = \ell}{\mathcal{V}, \eta \vdash \frac{t' + \texttt{Tpushvar}}{t'} \mid \frac{p'+1}{p'} \mid \frac{m}{m}\ x \rightsquigarrow \ell, \eta} \text{ (Variable)}$$

There are three rules for conditionals: two symmetric cases where the condition is true or false, respectively; and a third case to deal with exceptions (omitted here). In the case of a true/false condition the time cost is the cost of evaluating the conditional expression, plus the cost of evaluating an `If` instruction `Tiftrue`/`Tiffalse` plus the cost of executing the true/false branch, plus the cost of a goto if the condition is false.

$$\frac{\begin{array}{c}\mathcal{V}, \eta \vdash \frac{t_1}{t_1'} \mid \frac{p}{p'} \mid \frac{m}{m'}\ e_1 \rightsquigarrow \ell, \eta' \\ \eta'(\ell) = (\mathsf{bool}, \mathsf{ff}) \\ \mathcal{V}, \eta' \vdash \frac{t_1' - \texttt{Tiffalse}}{t_3'} \mid \frac{p'+1}{p''} \mid \frac{m'}{m''}\ e_3 \rightsquigarrow \ell'', \eta''\end{array}}{\mathcal{V}, \eta \vdash \frac{t_1}{t_3' - \texttt{Tgoto}} \mid \frac{p}{p''} \mid \frac{m}{m''}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \ell'', \eta''} \text{ (Conditional False)}$$

The Call rule deals with calls to some function fid, whether or not this is recursive. Each argument to the call is evaluated, and then the function is applied used App. The cost of the call is `Tcall` and the cost of completing the call is `Tslide`, where the underlying `Slide` instruction removes function arguments from the stack, whilst preserving the return value.

$$\frac{\begin{array}{c}\mathcal{V}, \eta_{(i-1)} \vdash \frac{t_{(i-1)}}{t_i} \mid \frac{p_{(i-1)}}{p_i} \mid \frac{m_{(i-1)}}{m_i}\ e_i \rightsquigarrow \ell_i, \eta_i \\ \mathcal{V}, \eta_k \vdash \frac{t_k - \texttt{Tcall}}{t_a'} \mid \frac{p_k}{p'} \mid \frac{m_k}{m'}\ \textsc{App}(fid, [\ell_k, \ldots, \ell_1]) \rightsquigarrow \ell, \eta'\end{array}}{\mathcal{V}, \eta_0 \vdash \frac{t_0}{t_a' - \texttt{Tslide}} \mid \frac{p_0}{p'+k} \mid \frac{m_0}{m'}\ fid\ e_k \cdots e_1 \rightsquigarrow \ell, \eta'} \text{ (Call)}$$

These rules can be easily extended to cover other expression forms and boxes, so giving a complete cost model for Hume. From this cost model, it is possible to derive a number of behavioural properties. The most important are that the cost model correctly captures the potential change in time and memory usage and that the result of execution is always left as an extra value on the stack. In order to produce this proof, we construct a formal translation from Hume to HAM, and prove for each case that the costs of the HAM translation are precisely captured in the cost model for the Hume source.

We have produced a prototype implementation of an analysis for space usage with non-recursive functions, based on this cost model [14], and calibrated against our abstract machine implementation. We are now working on extending the analysis to recursive functions and to include time information.

3 WCET Analysis using Abstract Interpretation

Our objective is to develop a combined high- and low- level analysis for worst-case execution time. We will achieve this by extending the stack and heap cost model presented above with the addition of parameters representing actual timing costs. Our ultimate aim is to produce accurate worst-case cost information from source level programs.

The AbsInt **aiT** tool (described below) uses abstract interpretation to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a given program point. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path.

The AbsInt analysis works at a code snippet level, analyzing imperative C-style code snippets to derive safe upper bounds on the worst-case time behavior. Whilst the AbsInt analysis works at a level that is more abstract than simple basic blocks, providing analyses for loops, conditionals and non-recursive subroutines, it is not presently capable of managing the complex forms of recursion which occur in functional languages such as Hume. We are thus motivated to link the two levels of analysis, combining information on recursion bounds and other high-level constructs from the Hume source analysis with the low-level worst-case execution time analysis from the AbsInt analysis.

3.1 WCET Prediction

Static determination of *worst-case execution time* (WCET) in real-time systems is an essential part of the analyses of overall response time and of quality of service [4, 18]. However, WCET analysis is a challenging issue, as the complexity of interaction between the software and hardware system components often results in very pessimistic WCET estimates. For modern architectures such as the Motorola PPC755, for example, WCET prediction based on simple weighted instruction counts may result in an over-estimate of time usage by a factor of 250. Obtaining high-quality WCET results is important to avoid seriously over-engineering real-time embedded systems, which would result in con-

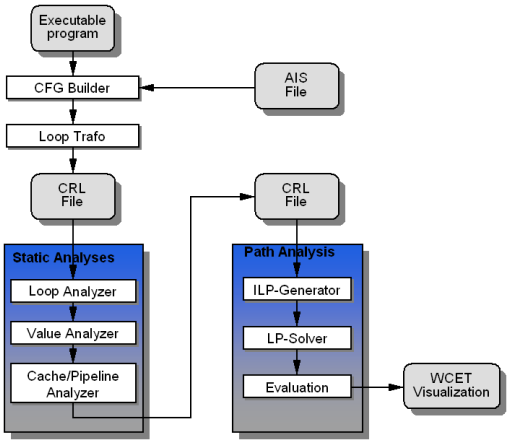


Figure 2: Phases of WCET computation

siderable and unnecessary hardware costs for the large production runs that are often required.

Three competing technologies can be used for worst-case execution time analysis: *experimental* (or testing-based) approaches [26], *probabilistic measurement* [2, 3] and *static analysis*. Experimental approaches determine worst-case execution costs by (repeated and careful) measurement of real executions, using either software or hardware monitoring. However, they cannot guarantee upper bounds on execution cost. Probabilistic approaches similarly do not provide absolute guaranteed upper bounds, but are cheap to construct and deliver more accurate costs than simple experimental approaches [2].

Motivated by the problems of measurement-based methods for WCET estimation, AbsInt GmbH has investigated a new approach based on static program analysis [16, 15]. The approach relies on the computation of abstract cache and pipeline states for every program point and execution context using *abstract interpretation*. These abstract states provide safe approximations for all possible concrete cache and pipeline states, and provide the basis for an accurate timing of hardware instructions, which leads to safe and precise WCET calculations that are valid for all executions of the application.

3.2 Phases of WCET Computation

In AbsInt's approach [9] the WCET of a program task is determined in several phases (see Figure 2):

- **CFG Building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from an executable binary program;
- **Value Analysis** computes address ranges for instructions accessing memory;
- **Cache Analysis** classifies memory references as cache misses or hits [10];
- **Pipeline Analysis** predicts the behavior of the program on the processor pipeline [16];
- **Path Analysis** determines a worst-case execution path of the program [22].

The cache analysis phase uses the results of the value analysis phase to predict the behavior of the (data) cache based on the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis, and pipeline analysis are all implemented using abstract interpretation [7], a semantics-based method for static program analysis. Integer linear programming is then used for the final path analysis phase.

The techniques described above have been incorporated into AbsInt's **aiT** WCET analyzer tools, that are widely used in industry [20, 5, 8, 27, 19]. For example, they have been used to demonstrate the correct timing behavior of the new Airbus A380 fly-by-wire computer software in a certification process according to DO178B level A [23, 21]. For this purpose, **aiT** for MPC755 and **aiT** for TMS320C33 will be qualified as verification tools according to DO178B.

3.3 Linking the Analyses

In order to link the two levels of analysis, we must base the costs for time potentials in the cost model (`Tpushvar` etc) on actual times for execution on the Hume Abstract Machine using information obtained from the **aiT** tool. In this way, we will have constructed a complete time cost model and analysis from Hume source to actual machine code.

Pragmatically, in order to obtain timing information from the **aiT** tool, our high level analysis must be adapted to output information on the limits on recursion bounds and other high-level constraints derived from the program source that can be fed to the **aiT** tool using its native system specification language (**aiS**). This information must be provided in terms of the compiled executable code that has been produced from the Hume source rather than directly from the source itself. It will

Cost	**aiT** bound for M32 (cycles)	Prob. bound for PPC (μs)
Tiftrue	30	0.051
Tiffalse	30	0.051
Tpushvar	109	0.110
Tmatchint	30. . .32	0.047
Tmatchedrule	11	0.039
Tmatchrule	22	0.053
Tmatchnone	11	0.040
Tconsumeset	82	—
Tmkint	220 . . . 223	0.046
Tcopyarg	110	0.045

Figure 3: WCET bounds on HAM instructions

therefore also be necessary to provide details of the compilation process in an appropriate form.

3.3.1 Preliminary WCET Results

This section reports timing results obtained using the **aiT** tool using the IAR C-compiler for the Renesas M32C. The M32C is a 32-bit architecture designed for typical automotive applications. It has a complex instruction set and a three-stage pipeline, but neither data nor instruction cache. Instruction cache analysis is therefore disabled. Figure 3 gives timings obtained from **aiT** for some sample HAM abstract machine instructions on the M32, and the corresponding costs on a 1.25GHz PowerPC G4 obtained using a probabilistic approach over 1,000,000 executions of each instruction (we have not yet been able to obtain probabilistic cost information for the M32, though we expect to be able to achieve this soon). While there are some clear differences in the underlying implementation of the instructions on the two architectures (notably for `Tmkint`, which allocates heap in external memory), there are also broad similarities.

An interesting observation is that combining the WCET costs of individual HAM instructions gives a result that is usually within 1-2% of the WCET bound of the complete sequence of instructions. While this observation certainly holds as long as the code on the M32C is executed from internal memory with single cycle access time, for slower, external memory, the internal instruction buffer of the M32C might have a bigger influence. This means that for "simple" architectures, WCET bounds for Hume-like languages can be computed by considering WCET bounds of individual abstract machine instructions.

4 Conclusions

We have introduced Hume and shown how a cost model can be constructed to expose time, stack and heap cost information. We have also outlined how our work can be extended in order to synthesise worst-case execution time costs using a combination of source- and binary-based analysis. Our work is formally based and motivated: we aim to construct formal models of behaviour at source program and abstract machine levels; have provided elsewhere a formal translation between these levels; and will synthesise actual worst-case execution time costs using abstract interpretation of binary programs.

References

[1] J. Armstrong, S.R. Virding, and M.C. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proc. 12th Euromicro Intl. Conf. on Real-Time Systems*, Stockholm, June 2000.

[3] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc 23rd IEEE Real-Time Systems Symp. (RTSS 2002)*, Dec 2002.

[4] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[5] Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Applying static WCET analysis to automotive communication software. In *17th Euromicro Conf. on Real-Time Systems, (ECRTS'05)*, July 2005.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. ACM Symp. on Princ. of Prog. Langs. (POPL '87)*, 1987.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Princ. of Prog. Langs. (POPL '77)*, pages 238–252, 1977.

[8] Ola Eriksson. Evaluation of static time analysis for CC systems. Technical report, Mälardalen University, August 2005.

[9] 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 *Proc. EMSOFT 2001, First Workshop on Embedded Software, Springer-Verlag LNCS 2211*, pages 469–485, 2001.

[10] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems, Saarland University, Saarbrücken, Germany.* PhD thesis, 1997.

[11] T. Gautier, P. Le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lect Notes in Computer Science*, pages 257–277. Springer-Verlag, 1987.

[12] K. Hammond. Is it Time for Real-Time Functional Programming? In *Trends in Functional Programming, volume 4*. Intellect, 2004.

[13] K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[14] K. Hammond and G.J. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. Implementation of Functional Langs. (IFL '02), Madrid, Spain*, volume 2670 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[15] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[16] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proc. Intl. Static Analysis Symp. SAS 2002, Springer-Verlag LNCS 2477*.

[17] G. Michaelson, K. Hammond, and J. Sérot. The Finite State-ness of Finite State Hume. In *Trends in Functional Programming, Volume 4*. Intellect, 2004.

[18] Johan Nordlander, Magnus Carlsson, and Mark Jones. Programming with Time-Constrained Reactions. http://www.cse.ogi.edu/pacsoft/projects/Timber/publications.htm. 2006.

[19] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[20] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Cyprus, October 2004.

[21] Daniel Sehlberg. Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, October 2005.

[22] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[23] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[24] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. 2003 IEEE Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 625–632, 2003.

[25] M. Wallace and C. Runciman. Extending a Functional Programming System for Embedded Applications. *Software: Practice & Experience*, 25(1), January 1995.

[26] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ACM Intl. Conf. on Functional Programming (ICFP '01)*, Sep 2001.

[27] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proc. IEEE Workshop on Software Tech. for Future Embedded and Ubiquitous Systs. (SEUS'05)*, pages 7–10, 2005.

[28] Yina Zhang. Evaluation of methods for dynamic time analysis for CC-systems AB. Technical report, Mälardalen University, 2005.

PapaBench : A Free Real-Time Benchmark

Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun and Marianne De Michiel
IRIT - University of Paul Sabatier
F-31062 Toulouse France
{nemer, casse, sainrat, bahsoun, michiel}@irit.fr

Abstract

This paper presents PapaBench, a free real-time benchmark and compares it with the existing benchmark suites. It is designed to be valuable for experimental works in WCET computation and may be also useful for scheduling analysis. This benchmark is based on the Paparazzi project that represents a real-time application, developed to be embedded on different Unmanned Aerial Vehicles (UAV).

In this paper, we explain the transformation process of Paparazzi applied to obtain the PapaBench. We provide a high level AADL model, which reflects the behavior of each system component and their interactions.

As the source project Paparazzi, PapaBench is delivered under the GNU license and is freely available to all researchers. Unlike other usual benchmarks widely used for WCET computation, this one is based on a real and complete real-time embedded application.

1. Introduction

When designing a real-time system, it is mandatory to have a predictable timing of the system. While underestimating the execution time of tasks may cause catastrophic disasters especially in critical hard real-time systems, overestimating the execution time may also cause an oversizing of the running hardware.

To prove these timing constraints, it is essential to know the Worst Case Execution Time (WCET) of a program running on a particular hardware system. The real-time system designers use it to check the timing deadlines satisfaction of the tasks while many real-time operating systems rely on this information to perform scheduling. Moreover, in embedded system design, the WCET of the software is often required in order to decide how to partition hardware / software.

As any piece of software, the WCET computation needs to be experimented, evaluated and compared. To achieve this goal, this paper introduces PapaBench, a real time benchmark, describing a complete embedded system driving a UAV. Designed to be a valuable base for experimental work in the WCET computation by static [1, 2, 3, 4, 5] or dynamic [6] analyses, it may be also very useful for scheduling analysis of applications since it provides concrete tasks and interrupts with their timing constraints and precedence rules. This benchmark will make experimental results more realistic than existing WCET benchmarks [7, 8] since the tasks encountered are close to those running in real avionic systems.

The rest of this paper is organized as follows. Section 2 provides a complete description of Paparazzi. Section 3 presents our PapaBench model in AADL [9, 10], which maps the Paparazzi C sources into a list of tasks and interrupts. Section 4 describes the PapaBench genesis, the adaptation to compile this benchmark on different architecture and the mapping of the application sources with the AADL model. We compare our benchmark with existing real-time benchmarks in section 5 and section 6 concludes this paper.

2. The Paparazzi Project

The "Paparazzi" project, created in 2003 by P. Brisset and A. Drouin [11, 12], is an attempt to build a cheap fixed-wing autonomous UAV executing a predefined mission. It develops a complete system hardware and software that may be installed on a variety of aircrafts. Such a system has limited autonomies, a 2-5 kg total aircraft weight, a 25 km maximum flight distance, a one hour flight duration, a 50 km/h maximum speed and a 500 g maximum payload.

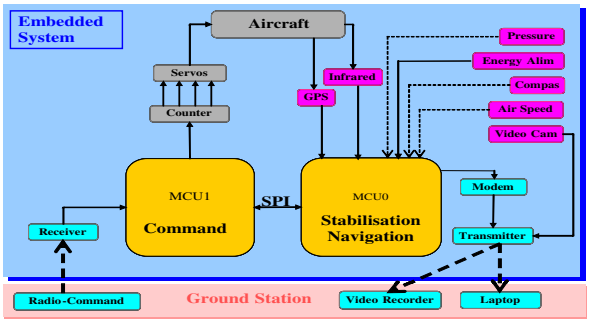


***Figure 1**: Paparazzi*

It comprises an embedded system and a ground station as shown in Figure 1. The embedded system consists of a control card, a GPS receiver (blox SAM-LS with 16 channels), a two-axis differential infrared sensor, a radio transmitter and servo-commands controlling gaz and wings. We have also a list of devices supplying voltage, pressure, heading and so on.

The control card is designed as a bi-processor architecture, separating the radio / servo commands management from the autopilot task, holding two RISC, ATMEL AVR micro-controllers [13]. MCU1 (ATMega8, called Fly-By-Wire) features a 16 MHz / 16 MIPS processor with 1 Kb SRAM, a 8 Kb flash memory and 512 bytes EEPROM, that manages radio-command orders

and servo-commands. MCU0 (ATMega128, called Autopilot) provides a 16 MHz / 16 MIPS processor with 4 Kb SRAM, 128 Kb flash memory and 4 Kb EEPROM. It runs the navigation and stabilization tasks of the aircraft. The micro-controllers are inter-connected by a SPI serial link in a master (MCU0) / slave (MCU1) mode.

The ground station consists of a usual radio-command, a radio receiver and a laptop. The audio and video channels of the receiver are respectively connected to the laptop and to a video tape recorder. The laptop receives information about the running mission while a variety of interfaces visualize the flight parameters, the flight path and all the messages held by the aircraft.

Although the ground station is largely developed, we are only interested in the embedded system that constitutes the core of our benchmark. Indeed, the ground station software does not exhibit any hard real-time code.

The embedded system has two basic operation modes: "manual" mode and "automatic" mode. In "manual" mode, MCU1 receives the radio-command instructions from the ground station and dispatches them to MCU0. MCU0 analyses this information, performs the stabilization and returns the flight commands to the MCU1 that transmits them to the servos.

On the other hand, in "automatic" mode, MCU0 manages the aircraft navigation using the GPS and the infrared sensor while MCU1 only receives the flight commands and transmits them to the servos. In this mode, the aircraft has a specific mission defined in a high level language. Thereby, there are three control levels: the mission, the navigation and the stabilization.

If MCU0, possibly crashed, sends no more commands and the radio-command is unreachable, the system switches to the failsafe mode: the engines are stopped and the aircraft glides to the ground.

In conclusion, Paparazzi is a realistic real-time embedded system exhibiting a quite complex behavior. Yet, unlike most equivalent industrial systems, the sources are freely available.

3. Modeling with AADL

Unlike other WCET benchmarks, PapaBench is close to actual running systems, rendering the experimentation results more realistic and making it possible also to handle real effects of task chaining. In order to cope with the Paparazzi embedded system, we have first produced an AADL model describing the whole system. Unlike other benchmarks, PapaBench is not a collection of independent programs but provides a full application. Consequently, we need a way to split it in tasks and to model the dynamic behavior of the system: AADL is widely used in avionics field to achieve this goal.

3.1. About AADL

We have decided to depict this application in AADL (Architecture Analysis and Design Language) because it is a formal specification of real-time embedded, fault-tolerant, securely-partitioned, dynamically-configurable systems. It covers the domain of distributed multiple-processor hardware architectures as found in avionics, robotics and automotive.

A system modelled in AADL consists of an application software mapped to an execution platform. It describes how components are combined into subsystems, how they interact and how they are allocated to hardware components.

AADL has ten basic component's types divided into three categories: software, hardware and composite. Data, thread, thread group, process and subprogram constitute the first category. The hardware category holds processors, memories, buses and devices. The “system” is the only composite element.

3.2. AADL Usage

An AADL model depicts the overall application with an accurate model of the whole embedded system. As AADL provides a textual and graphical view of the system, the user can easily understand the internal application work.

Moreover, the well-defined AADL language and its openness may be used to perform automatic processing. For example, different schedules may be generated from the description: in our experimentation, we plan to use CHEDDAR [14] for this task. In particular, we plan to use the scheduling results and the AADL model of the application to analyse the WCET of a whole application running cycle. This analysis may be used to evaluate the full system workload or to handle hardware dependencies between tasks in order to improve the WCET accuracy.

Finally, although the model is based on a real application and while we do not perform functional simulation, we have some freedom to change it according to our experimentation needs. We may add / delete components, change components properties and / or add new properties to evaluate application parameters. As we are especially interested in timing constraints in WCET and scheduling analysis, an AADL model can be very useful to evaluate these properties without having to change the application structure.

3.3. Paparazzi AADL Model

We found the different control levels and the corresponding timing constraints in the report on the Paparazzi project [11].

Based on this reference, and after analyzing the C files of the embedded software, we have identified a list of tasks executed in this application as well as their timing constraints and their precedence rules. We have also determined the interrupts used to drive the hardware. Table 1 on next page shows the tasks and the interrupts executed by MCU1. Table 2 displays those treated by MCU0. We provide for each task and interrupt an identifier used in the following section, a description and the appropriate frequency.

The precedence rules that sets an order on tasks execution are depicted as a graph in Figure 2. This order is required by data dependencies (edges marked with 1) or

ID	Description	Frequency
T1	Receive Radio-Command orders	40Hz
T2	Send Data to MCU0	40Hz
T3	Receive MCU0 values	20Hz
T4	Transmit Servos	20Hz
T5	Check Failsafe	20Hz
I1	Transmission Servos interrupt	-
I2	SPI interrupt for MCU1	-
I3	interrupt Radio	-

Table 1: *MCU1 tasks and interrupts*

ID	Description	Frequency
T6	Managing Radio orders	40Hz
T7	Stabilization	20Hz
T8	Send Data to MCU1	20Hz
T9	Receive GPS Data	4Hz
T10	Navigation	4Hz
T11	Altitude Control	4Hz
T12	Climb Control	4Hz
T13	Reporting Task	10Hz
I4	SPI interrupt of MCU0	-
I5	Modem interrupt	-
I6	GPS interrupt	-

Table 2: *MCU0 tasks and interrupts*

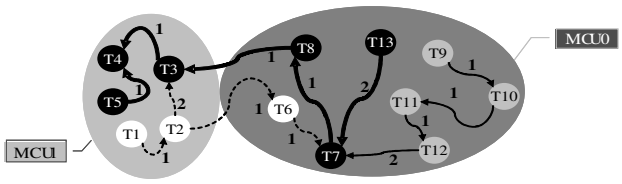


Figure 2*: Precedence Rules Graph*

control dependencies (edges marked with 2). The dashed arrows reflect the precedence rules valid in manual mode, the plain arrows represent the precedence rules in automatic mode and the thick ones are valid in both modes. The white circles reveal tasks executed only in manual mode, the gray circles are tasks executed in automatic mode and the dark ones are executed in both modes.

In manual mode, MCU1 receives information from the radio command (T1) and transmit it to MCU0 (T2). MCU0 executes T6, T7 and T8 to analyze radio-command instructions, to perform stabilization and to return the flight commands. Then T3 and T4 occurs to receive data from MCU0 and send it to servos. T4 enables the interrupt I1 to send information to the servos. This scenario persists as long as the system stays in this mode.

The automatic mode is activated by a radio command order or when the radio command is no more reachable. AADL provides modes to record the system operational mode. Variation in operational modes is triggered by events.

In automatic mode, T9 analyses the messages held by the GPS. The navigation (T10) is realized at the same frequency as the GPS information delivery, it also controls the mission to consider exceptional events. In this mode T10, T11 and T12 occur always in this order before the stabilization task T7. They cannot execute separately. They are followed by [T7, T8, T3, T4]. Note that the repetition of [T7, T8, T3, T4] is constrained by their period which is less than the period of T10, their execution time and the execution time of the previous tasks : these tasks may be viewed as the feedback control loop that must satisfy the navigation commands. In both modes, MCU0 reports changes in the aircraft path, in the operational mode, in the navigation and so on to the ground station in a task executed at 10 Hertz.

At the hardware level, the Paparazzi system has two subsystems, one for each micro-controller. Each system describes the execution process as a set of threads defining the tasks and interrupts executed in its context, the list of devices and the relations between components.

3.4. Variable Complexity

In AADL, each component type can be characterized by a set of properties. To include the timing constraints in our model, we added for each thread the period property and the dispatch protocol that can be either aperiodic, sporadic or periodic. An aperiodic task occurs at arbitrary times but can be delayed for a limited time, while a sporadic one occurs at irregular intervals with a maximum or minimum period between two consecutive executions. The properties of the components can be changed or extended in order to reflect the user demands.

The inputs / outputs in Paparazzi are managed as aperiodic interrupts. Usually, the avionics software does not support interrupts because the WCET cannot be accurately computed with the current techniques. However, we are not compelled to use the static scheduling provided in the Paparazzi C code. One may consider the AADL model, its tasks (including interrupts) and its matching code in C sources but different scheduling and timing properties may be experimented according to our needs. This leads us to define several models, from the simplest one to the most realistic one.

We can begin to work with the simpler configuration of the system, assuming that all tasks and interrupts are periodic. Then, we may improve our analysis toward more complex configurations, close to the real application behavior. In the AADL model, it is easily achieved by varying the “Dispatch_Protocol” property value in the set: “periodic”, “sporadic” and “aperiodic”.

We can also consider the preemption between tasks. For this purpose, we extended AADL with a new property, called “Preemption”, only applicable to the threads. This property indicates the preemption type that may be one of “System_Preemption”, “Time_Sharing_Preemption” or “Non_Preemptive”. We must also choose a preemptive scheduling protocol for the processor. This new property offers a framework for many kind of studies including WCET computation.

As an example of use of these different levels of complexity, we plan to experiment an approach allowing the WCET computation for a complete cycle of the application. We intend to begin our WCET analysis with the basic level where we will consider periodic tasks and periodic interrupts with no preemption, in order to define a possible schedule with the scheduler.

4. PapaBench Genesis

Source code, schematics and documentation of the Paparazzi project [12] are freely released under the GNU license. This section explains our analysis of Paparazzi, the transformation process to obtain PapaBench and the changes required to compile the benchmark.

4.1. System Instantiation and Restriction

The Paparazzi distribution is only available for a Linux environment but may be configured for several aircraft configurations. As we are not interested in the details of the hardware control, PapaBench is only bound to the default aircraft configuration. It includes a MC3030 radio-command, a Twinstar3 model-making aircraft, the flight plan used during the first European MAV Flight Competition held in Braunschweig Germany on July 13 2004 and a classic ground station as described in section 2.

The first generation of the embedded system enables us to save C header files, generated from XML sources. These XML files contain the configuration of the airframe, the radio commands and the flight plan that constitutes the mission. They make the Paparazzi project applicable to many different aircrafts and allow to realize different flight plans. After the generation, we saved the generated header files and included them in the benchmark without having to preserve the XML sources.

Then we analyzed the static sources and the generated headers in order to create PapaBench. We have excluded the sources of the ground station in charge of monitoring the flight, display statistics, programming the mission and generating the embedded system sources: they are composed of a mixture of OCAML and Perl programs not really involved in the embedded system.

4.2. Mapping the AADL Model

Using the OTAWA project [15, 16], a framework to experiment WCET computations and binary static analyses, we have developed a program generating the Program Call Graph (PCG) of an executable file and some other statistics about the executed binaries.

The PCG gives a general idea of the complexity of the application and it enables the user to have a clear view of the function call chains without reverting to the sources.

They also provide a map of the tasks identified in the logical analysis of the system to the matching implementation in the C sources.

The PCG of Fly-By-Wire and Autopilot are represented in Figures 3 and 4 respectively. The grey ellipses represent task implementation code. The identifier of this task is marked in a dark label.

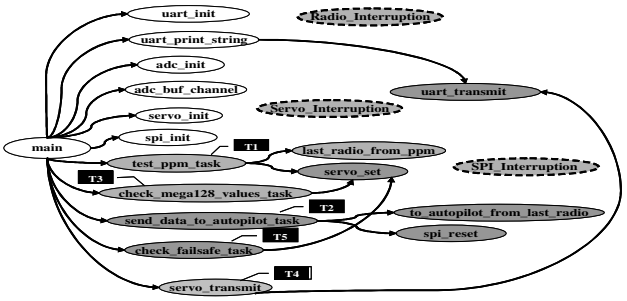


***Figure 3**: Fly-By-Wire PCG*

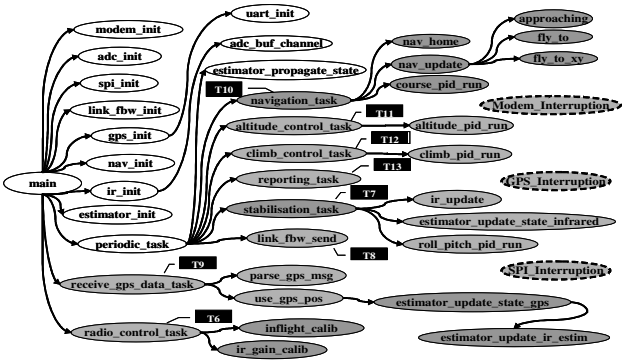


***Figure 4**: Autopilot PCG*

In the Autopilot PCG, T10 is course_pid_run and either nav_home if the system is in failsafe mode, or nav_update in automatic mode. The dashed ellipses show interrupts interfering with the execution of the subsystems tasks. One may notice that some subprograms are not part of any task: this code is only called at startup time and is not involved in the system execution during the flight.

4.3. Compilation Details

Our objective was to enable a user to compile the benchmark, without the requirement of the whole Paparazzi building environment, for different architectures. Until now, we have experimented the compilation for PowerPC and x86 architectures using the GCC compiler suite but it should be easy to adapt PapaBench to other configurations.

To compile the benchmark, one must extract the archive and edit the default configuration files, in the conf/ directory, to change the top directory path and the compiler command according to the target architecture. A simple call to make in the distribution top directory should compile everything.

It is important to mention that PapaBench includes headers files from the AVR C libc library project containing macros providing access to the AVR hardware registers like IO ports, timers, and so on. As the benchmark does not target hardware simulation, either the hardware registers only matter by their temporal properties, or they may be simply considered as simple memory accesses.

As they are mapped to low addresses (between 0x20 and 0x100), this might be impeding for some platforms where the addresses of interrupts vectors appear at this location. Fortunately, we can get rid of this problem by assigning in the compilation flags a compatible value to

the SFR_OFFSET definition, which is the base of the hardware register addresses.

5. Comparison with Other Benchmarks

Benchmarking constitutes a critical part of the design process. As real applications are not easily available to researchers due to the confidentiality criteria surrounding the industrial estate, real-time benchmarks are rare and often disconnected from the surrounding particularities of real-time systems. This section gives an overview of these benchmarks and compare them to PapaBench.

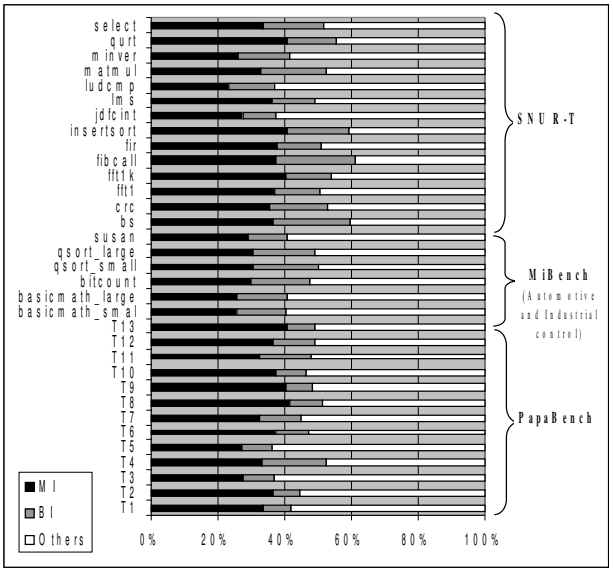
5.1. Other Benchmarks

Real-time benchmarks are usually a collection of basic algorithms found in real-time systems.

MiBench [7], for example is a set of 35 embedded applications divided into six suites, each one targeting a specific area of the embedded market. The six categories are 1) automotive and industrial control, 2) consumer devices, 3) office automation, 4) networking, 5) security and 6)telecommunications. All the programs are available in standard C source code and are portable to any platform that has compiler support. Some modifications has been made to the source to promote the portability of the benchmark and the extensibility of the data set. Where appropriate, MiBench provides a small and large data set. The small data set represents a light-weight, useful embedded application of the benchmark, while the large data set provides a real-world application. This benchmark has many similarities to the EEMBC suite as described on their website [17] but MiBench is composed of freely available source code. We only compare the category (1) of MiBench suite with PapaBench because other categories are not used in hard real-time systems.

The SNU Real-Time Benchmarks suite [8], consists of C sources collected from numerical calculation programs and DSP algorithms as binary search program, fast Fourier transform, Fibonacci series function, insertion sort, square root calculation, matrix multiplication and many other programs. The benchmarks have the following structural constraints: no unconditional jumps, no exit from loop bodies, no *'switch'* statement, no *'do...while'* construct, no multiple expressions joined by *'or'*, *'and'* operations and no library calls. These restrictions are caused by the limited capabilities of the compiler involved in the experimental analysis environment used by the benchmark creators.

The benchmarks mentioned above are disconnected from the surrounding particularities of real time systems. Their functions found are executed alone out of the context of a real application. On the other hand, PapaBench tasks are embedded in a real system with hard timing constraints. This feature allows the analysis of effects of tasks on the execution of other ones. It is worthy to use such an application because of its similarities with the industrial real-time applications.



MI(Memory instruction rate), BI (Branching Instructions)

Figure 5: *Instruction repartition*

	R_B	R_M	AS_{BB}	MS_{BB}
PapaBench	0,093	0,383	7,06	137
MiBench ***(Automotive & industrial control)***	0,181	0,275	4,63	150
SNU R-T	0,15	0,341	5,27	89

R_B (Branching rate), R_M (Memory access rate), AS_{BB} & MS_{BB} (average & maximum size of Basic Blocks)

Table 3*: Statistics*

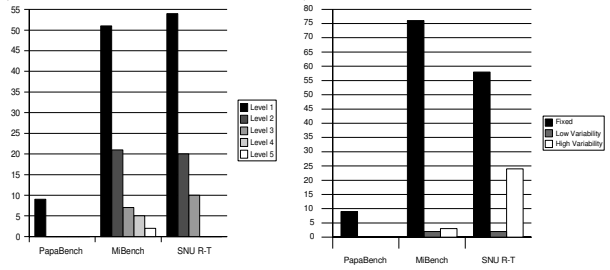
5.2. Code Characteristics

We have used the OTAWA framework to characterize the PapaBench code as well as MiBench and SNU R-T codes. Tables 3 gives, for each benchmark, the branching rate, the memory access rate, the average size and the maximum size of basic blocks. Figure 5 provides, for each task of the benchmarks, the rate of memory accesses (black area), of branching instructions (gray area) and of other instructions.

First, we can see that PapaBench has small basic blocks except for T1 and T9 where the maximum size of basic blocks is 137 and 110 respectively: it seems that the radio management and GPS data analysis require a lot of computations. SNU RT functions also have small basic blocks except for jdfcint and MiBench_Automotive has big basic blocks for the majority of its tasks : MiBench is too much oriented toward computations unlike the other benchmarks.

We found a high level rate of memory accesses in the three benchmarks which reflects the importance of the memory hierarchy analyses in the WCET computation. While, in case of PapaBench, it is caused by lots of hardware register accesses, other benchmarks seem to have a too big memory foot print.

To sum up, PapaBench has some similarities with other real-time benchmarks as they all have close memory access and branching rates. This also confirms that these benchmarks are close to real applications. Next section will show that differences exist and these statistics are not



(a) Loops Depth *(b) Loop Bound Variability*

***Figure 6**: Loop Complexity Analysis*

enough to characterize a benchmark.

5.3. Loop Complexity

CFG and syntactic tree representations are not enough for static WCET computation since they don't identify bounded execution paths. Hence these representations have to be completed by some information to restrain the number of executable paths to consider in the analysis. In this paragraph, we discuss the PapaBench loop complexity and compares it with the other benchmarks.

Graph (a) of Figure 6 displays for each benchmark, the loops repartition among 5 nested levels. The darker column represents top level loops counts and the columns get brighter as the level is deeper. Moreover, graph (b) reflects the variability level of loops maximum iteration numbers. We distributed benchmarks loops over three levels: 1) for loops with fixed iteration number, 2) for loops with little variation in the iteration number (depending on parameters with a constant during the function call) and 3) for loops with high variability degree (induced by loops nesting with an inner loop bound depending on the outer loop induction variable).

The loops encountered in PapaBench are mostly *for* loops, we have only two *while* statements. The *for* loops maximum iteration number is fixed but the *while* loops analyses gives a maximum iteration number of 0 or 1. Thus, we mostly do not have variations in loop bounds. Moreover, we can notice that PapaBench loops are simple with no nesting as shown in graph (a). On the other hand, MiBench and SNU-RT benchmarks contain different nesting levels of loops with variable iteration numbers. A high level of variability makes WCET analyses more complicated or increases the approximation pessimism since the user have to provide an upper bound of loops iterations. If the loop bound is not represented in a fine way (as constants for example), the real number of iteration may be over-estimated due to loops nesting and the variability of loops bounds. However, the PapaBench case and our experience in avionics software show that we have more often simple loops with a fixed number of iterations. This makes WCET calculation accurate and closer to the real WCET. In the other hand, it seems that other benchmarks exhibit over-complicated program structures.

6. Conclusion

Benchmarking is a critical problem in WCET computation because real applications are not easily available due to the confidentiality criteria surrounding the industrial estate. In this paper, we introduced PapaBench is a complete real-time embedded application derived from a real application used to control a UAV. This prominent feature makes it mostly useful in WCET and scheduling analyses and unique among the other existing benchmarks. We have given a whole description from the system point of view, using an AADL model, and an instruction level analysis. We have also compared it with existing real-time benchmarks to denote similarities and advantages that makes it useful and unique in WCET computation domain.

In the near future, we plan, to perform new analyses of the PapaBench AADL model: we will consider two levels of complexity, for the periodicity of tasks and interrupts. These restrictions will be used to validate a WCET computation approach based on the whole application cycle. Note that PapaBench sources and AADL model are available at http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=22.

7. References

[1] Y.-T. S. Li, S. Malik. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. 16th IEEE Real-Time Systems Symposium, pages 298-307, December 1995.

[2] Y.-T. S. Li, S. Malik. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. 17th IEEE Real-Time Systems Symposium, 1996.

[3] C. Ferdinand et al. Applying Compiler Techniques to Cache Behavior Prediction. ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems: 37–46.

[4] F. Muller. Generalizing Timing Predictions to Set-Associative Caches. Technical Report TR 96-66, Institut für Informatik, Humboldt-University, July 1996.

[5] Y. Tan, V. Mooney. Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-Tasking Real-Time Systems. 8th International Workshop, SCOPES 2004, in Lecture Notes on Computer Science, LNCS3199, pages 182–199, 2004.

[6] I.Wenzel, B.Rieder, R. Kirner, P. Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. Design, Automation and Test in Europe, Volume 1, pages 606-611. March 2005.

[7] M.R. Guthaus, J.S. Ringenberg. Austin, T. Mudge and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. 4th Workshop on Workload Characterization, Dec. 2001, Austin, TX.

[8] SNU Real-Time Benchmark Suite. http://archi.snu.ac.kr/realtime/benchmark.

[9] P. Feiler, D. P. Glush, J. J. Hudak, B. A. Lewis. Embedded System Architecture Analysis Using SAE AADL, June 2004.

[10] SAE International. Architecture Analysis & Design Language (AADL), August 2004.

[11] P. Brisset. Drones civils perspectives et réalités. Technical report, Ecole nationale de l'aviation civile, August 2004.

[12] www.recherche.enac.fr/paparazzi

[13] ATMEL Corporation. ATMega128 complete datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[14] F. Singhoff, J. Legrand, L. Nana. AADL resource requirements analysis with Cheddar. LYSIC/EA 3883.

[15] H. Cassé, C. Rochange, P. Sainrat. An open Framework for WCET Analysis. IEEE Real-Time Systems Symposium-WIP session, pages 13-16, Lisbon, December 2004.

[16] H. Cassé, C. Rochange, P. Sainrat. OTAWA, a framework for experimenting WCET computations. 3rd European Congress on Embedded Real-Time, Toulouse, December 2005.

[17] EEMBC Real-Time benchmarks . http://www.eembc.com.