T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 5.5 Report on Coding Policies for Time-Predictability

Version 1.0
7 February 2014
Final

Public Distribution

Vienna University of Technology, AbsInt Angewandte Informatik, GMVIS Skysoft, Intecs

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft José Neves Av. D. João II, Torre Fernão de Magalhães, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: jose.neves@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First outline.	15 November 2013
0.2	Input-Data Dependency Analysis	13 December 2013
0.3	Coding Policies for Analyzability	20 December 2013
0.9	Pre-Final version, requesting partner comments	31 January 2014
1.0	Final version	7 February 2014

Executive Summary

This document describes the deliverable *D5.5 Report on Coding Policies for Time-Predictability* of work package 5 of the T-CREST project, due 27 months after project start as stated in the Description of Work. The deliverable describes the guidelines and restrictions on coding to produce code with stable and analysable timing that performs well in the worst case. We study existing software development guidelines that are currently in production use and identify coding rules that might ease a static timing analysis of the developed software. Additionally, we present a novel coding policy that restricts the dependencies of control-flow decisions on input data and ensures that all loops are analysable by means of static analysis techniques.

1 Introduction

Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. Experience with the use of static timing analysis methods and the tools based on them in the automotive and the avionics industries is positive. However, both, the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform [4] and of the software run on this platform.

In this report, we concentrate on the effect of the software on the time predictability of the embedded system. We study existing software development guidelines that are currently in production use and identify coding rules that might ease a static timing analysis of the developed software. Such coding guidelines are intended to lead the developer to producing – among others – reliable, maintainable, testable/analysable, and reusable software. Code complexity is also a key aspect due to maintainability and testability issues. However, existing coding rules are not explicitly intended to improve the software predictability with respect to static timing analysis.

In the first part of this report, we present a novel coding policy that does not restrict the developer by disallowing programming idioms but instead requires that certain control-flow decisions must not depend on input-data. This restriction is only meaningful if a sound and precise definition of input-data dependence is used, which is developed in the course of this report. The resulting coding policy is easy to enforce, even in a modular development process, and guarantees that loop bounds are always found by means of static analysis.

In the second part, based on our experience of analysing automotive and avionics software, we provide additional means to increase software time predictability. Certain information about the program behaviour cannot be determined statically just from the binary itself (or from the source code, if available). Hence, additional (design-level) knowledge about the system behaviour would allow for a more precise (static) timing analysis. For instance, different operating modes of a flight control unit, such as *plane is on ground* and *plane is in air*, might lead to mutual exclusive execution paths in the software system. By using this knowledge, a static timing analyser is able to produce much tighter worst-case execution time bounds for each mode of operation separately.

The rest of this document is structured as follows. Section 2 presents coding policies that allow automated path analysis. In Section 3 we discuss coding policies that improve the analysability of tasks. In Section 4 we briefly summarise existing practises from our industry partners. Section 5 documents how we met the WP 5 requirements, and Section 6 concludes this report.

2 Coding Policies for Automated Path Analysis

In safety-critical real-time systems, we need sound and precise predictions of the system's timing behaviour to verify its correctness. In order to ensure the final system will meet its deadlines and to reduce the cost of certification, it is highly desirable to perform automated analyses routinely during development. Existing guidelines from the safety-critical domain already mandate that it is possible to run certain analyses automatically on production code [14]. For time-critical systems, the development tool chain thus should be able to perform Worst-Case Execution Time (WCET) analyses in an automated way, and provide feedback to the developers of the system.

One approach to ensure analysability is to use the WCET analysis tool as a decision procedure that is run at regular intervals and determines whether a program fragment is analysable. If the WCET analysis tool is able to derive a (sufficiently precise) WCET bound, the software is taken to be analysable. If the analysis fails, the feedback of the analysis tool (i.e., error messages) is used to improve the analysability of the system. This strategy needs no additional tool support, but appears to be inadequate in a modular development process. On the one hand, the provider of a module or library is usually not aware in which ways the software provided to the client will be used, and thus can only ensure the analysability of certain usage scenarios. On the other hand, the client of the library might not be aware of the preconditions that need to be met in order to ensure that the functions provided by the library are analysable, or how to meet these preconditions. Due to this limitations, we are interested in alternative approaches that provide direct guidance to ensure the analysability of the system.

There are several software guidelines for safety-critical software that forbid certain constructs known to be problematic, such as loop bounds depending on the comparison of floating point variables [1]. These guidelines are certainly helpful, but do not offer constructive advise either. A draconian restriction that ensures that WCET analysis can be automated is to require a constant iteration count for all loops, and forbid recursive function calls. In theory, it is always possible to construct real-time tasks in this way, as all loops need to be bounded by a constant in order to calculate a WCET bound. This restriction however inhibits code reuse, as it is common that library functions feature loop bounds that depend on parameters provided by the client. For example, the loop bound of a routine that copies bytes from one memory area to another, or of a routine that searches for a key in an array, will vary between different call contexts.

The starting point for our approach is the observation that the control-flow of single-path code [22] can be analysed in a precise and automated way, by simply observing the execution trace. The characteristic property of single-path code is that all control-flow decisions are input-data independent, and thus do not depend on the state of the program or the system's environment. This observation suggests that investigating input-data dependencies might provide the right guidance for the construction of analysable systems. The following questions are of particular interest in this respect:

1. What is an sufficiently precise definition of input-data dependency that permits modular analysis?

2. Is there a code property based on data dependencies that is less restrictive than single-path, but still guarantees that automated control-flow analysis is feasible?

```
constant c;
inputs i1,i2;

if(i1) { x = c; f(x); }
else   { x = i2; g(x); }

h(x);
```

(a) Source

B_1: $p = \neg i_1$
br p, B_3, B_2
B_2: $x_1 = c$
f(x_1)
br B_4;
B_3: $x_2 = i_2$
g(x_2)
B_4: $x_3 = \phi(B_2 \rightarrow x_1, B_3 \rightarrow x_2)$
h(x_3)

(b) SSA Representation

B_1: p $= \neg i_1$
br p, B_3, B_2
B_2: $x_1 = c$
f(x_1)
br B_4;
B_3: $x_2 = i_2$
g(x_2)
B_4: $x_3 = \gamma(x_1, x_2; \begin{cases} x_2 \text{ if } p \\ x_1 \text{ if } \neg p \end{cases})$

(c) TGSA representation

Figure 1: Example illustrating input-data dependencies in an acyclic CFG (a) Different definitions of x should be distinguished, because the argument of g is input-data dependent, where as the argument of f is not (b) The SSA representation distinguishes different definitions of x, but does not model the control dependency between x_3 and i_1 (c) In the TGSA representation, the dependency between x_3 and i_1 is made explicit in the argument of the γ instruction

3. What preconditions are sufficient to ensure that the compiler is able to generate single-path code? Is it possible to ensure these conditions in a modular development process?

In the following, we extend our previous work [15] to address these questions. We start by deriving a precise definition of input-data dependencies, and an algorithm to compute these dependencies.

2.1 Precise Input-Data Dependencies

The purpose of an input-data dependency analysis is to decide which variables depend on values that are not in control of the analysed program, and to identify control-flow decisions that do not depend on input data. Input-data dependent values include values obtained from the environment (e.g., by sensors) as well as values that depend on the timing and behaviour of other tasks (e.g., shared variables). Due to the inherent limitations of static analysis techniques, it is necessary to approximate the property of being input-data dependent for practical purposes. In particular, we will not consider independence due to the semantics of arithmetic operations in this work. For example, if `z` is defined as by a statement `z = input * (y & 1)` , we always assume that `z` depends on `input`, although this is not necessarily the case if the set of possible values for `y` is restricted.

A modular way to model input-data dependence is to define a dependency relation between variables. It is advisable to distinguish distinct definitions of variables that share the same name (Figure 1a). The most straight-forward way to achieve this is to use a program representation in Static Single Assignment (SSA) form, where every variable is defined exactly once (Figure 1b).

We focus on the intra-procedural problem without access to global memory first. In the absence of loops, each variable is assigned exactly once during execution, and thus the value assigned to the variable is determined by some of the procedure's inputs and parameters. This observation suggests that it should be possible to compute the value assigned to a variable if all of its dependencies are known. In the presence of loops, the right choice for the dependency definition is less obvious, as

```
sensor input i;

// read from sensor
unsigned x = 0;
unsigned y = sensor();

// exchange x and y
while(y > 0) {
  x = x + 1;
  y = y - 1;
}
z = x;
```

(a) Source Code

B_0: $y_0 = i$;
$x_0 = 0$;
B_1: $x_1 = \phi(B_0 \rightarrow x_0, B_2 \rightarrow x_2)$
$y_1 = \phi(B_0 \rightarrow y_0, B_2 \rightarrow y_2)$
$c = y_1 > 0$
br c, B_2, B_3
B_2: $x_2 = x_1 + 1$
$y_2 = y_1 - 1$
br B_1
B_3: z = $\phi(B_1 \rightarrow x_1)$

(b) SSA Representation

B_0: $y_0 = i$;
$x_0 = 0$;
B_1: $x_1 = \mu(x_0, x_2)$
$y_1 = \mu(y_0, y_2)$
$c = y_1 > 0$
br c, B_2, B_3
B_2: $x_2 = x_1 + 1$
$y_2 = y_1 - 1$
br B_1
B_3: z = $\eta(x_1 ; c)$

(c) TGSA representation

Figure 2: This example illustrates input-data dependencies and loops (a) Note that the final value of x only depends on the initial value of y (b) The SSA representation is loop-closed: the definition of z in the exit block B_3 uses a ϕ node to explicitly merge the variable x_1 leaving loop B_1 (c) The value of z is determined by the number of times the loop B_1 is executed. This dependency is made explicit in TGSA form, in the third argument of the η node

different values might be assigned to a variable in different loop iterations. Therefore, the value assigned to a variable inside a loop not only depends on inputs, but might also depend on the current iteration count of one or more loops that include the definition of the variable. The number of times a loop is iterated in turn depends on certain conditions that are computed in each loop iteration. For example, in Figure 2, the value assigned to x_1 depends on the current loop iteration, but not on any input. The value of z, however, and hence the maximal value of x_1, depend on the number of times the loop body is executed, and in turn on y_0 and i. In summary, the dependencies relation should be chosen in such a way, that the value assigned to some variable is determined by the current loop iteration count and the value of its dependencies prior to the assignment.

2.1.1 Static Single Assignment Form and Program Dependence Graphs

We now turn to a formal definition of input-data dependencies, and suitable program representations that capture these dependencies. In SSA form, so called *def-use* dependencies are represented in an explicit way. If x_{n+1} is defined as $x_{n+1} = f(\ldots, x_i, \ldots)$, then x_{n+1} depends on the definition of x_i, and, by transitivity, on all variables x_i depends on. The example in Figure 1 illustrates that def-use dependencies are not sufficient to capture input-data dependencies, however. Although the value assigned to x_3 clearly depends on i_1, the transitive closure of the def-use dependency relation does not include this dependency. Therefore, while the SSA representation is a suitable starting point, additional information is needed to take to take dependencies between conditional branches and ϕ nodes into account.

A well-known program representation that accounts for control-flow dependencies is the Program Dependency Graph (PDG) [8]. It has been successfully used for program slicing, a technique that removes parts of the program that do not influence a property of interest. In the PDG representation, a

definition is *control dependent* on a branch condition if the condition influences whether the definition might be executed. More precisely, a statement in a basic block B_j is control dependent on the conditional branch defined in basic block B_i if

1. there is one successor of B_i that is post-dominated by B_j (the corresponding edge forces the execution of B_j)
2. there is one successor of B_i that is not post-dominated by B_j (the corresponding edge may avoid the execution of B_j)

Equivalently, B_j is control dependent on B_i if B_i is in the post-dominance frontier of B_j [6].

While control dependencies have their applications, we argue that they are not a suitable formalism for precise input-data dependencies. According to definition of control dependencies, a variable depends on all control-flow decisions that influence whether the definition might be executed. Control dependencies do not take the value assigned to a variable into account, and consequently it is ignored whether the control dependencies of a variable have any influence on its value at runtime. In the example from Figure 1, x_1 has a control dependency on the condition of the branch, and hence on i_1, although the value assigned to x_1 is a constant. For this reason, we propose a different definition, based on a gated program representation.

2.1.2 The Thinned Gated Single Assignment Form

The input-data dependency analysis presented here builds on a variant of the TGSA program representation [12]. Gated representations such as TGSA have been used to perform program analysis by means of backward substitution, a problem that is conceptually similar to chasing dependencies [28]. The TGSA representation can be realised as an extension of the Loop-Closed SSA (LCSSA) representation that is already available in the LLVM compiler framework. The LCSSA representation is an SSA program representation where all loops are natural and in canonical form. That is, for every loop there is a unique preheader, header and latch. Additionally, the target of exit edges (called exit blocks) always have exactly one predecessor inside the loop. In LCSSA form, every variable that is defined inside a loop is used only in well-defined instructions outside the loop, namely as the only argument to a ϕ node located in an exit block of the loop (loop-closed property). For reducible Control-Flow Graphs (CFGs), the properties required by the LCSSA form are easily established by standard program transformations; in the presence of irreducible loops, more sophisticated preprocessing steps (such as case splitting) are necessary.

In our TGSA representation, there are three distinct types of instructions that correspond to ϕ nodes:

μ nodes The instruction $\mu(x_1, x_2)$ is always defined in the header of a loop L, and takes two variables as arguments. The first argument needs to be defined outside the loop L, whereas the second argument needs to be defined in a member of L. Semantically, a μ node evaluates to the value of the first argument in the first iteration of the loop. In subsequent iterations, it evaluates to the value of the second argument in the previous loop iteration. In LCSSA form, header nodes have exactly two predecessors, the preheader defined outside the loop, and the latch defined inside the loops. Therefore, all ϕ nodes located in loop headers correspond to μ nodes.

η nodes The instruction $\eta(x_1, E[p_1, \ldots, p_n])$ is always defined in an exit block of the loop L that defines x_1. The η instruction selects the value assigned to x_1 in the last loop iteration. The number of loop iterations of L is determined by the expression $E[p_1, \ldots, p_n]$, that references predicates p_1 through p_n defined inside L. The expression E evaluates to `true` in the last iteration of L, and to `false` in previous iterations. The expression E is represented as a decision diagram, a Directed Acyclic Graph (DAG) where inner nodes correspond to decisions, and leaves correspond to values the expression may evaluate to. In LCSSA form, all ϕ nodes in exit blocks correspond to η nodes.

γ nodes The instruction $\gamma(x_1, \ldots, x_n; D[p_1, \ldots, p_n])$ selects one of the reaching definitions x_1 to x_n, depending on the expression $D[p_1, \ldots, p_n]$. Whereas in SSA form, the last executed basic block determines which definition is selected, in the TGSA representation this selection is determined by the values of the Boolean variables p_1 to p_n. The last argument $D[p_1, \ldots, p_n]$ is represented as a decision diagram. The inner nodes of the decision diagram are the branch conditions p_1 through p_n, and its leaves reference the reaching definitions x_1 to x_n. Semantically, the γ node evaluates the decision diagram D, and returns the value of the matching reaching definition x_i. Starting from the LCSSA representation, all ϕ nodes that are neither μ nor η nodes are replaced by γ nodes. Note that all Boolean variables p_i need to be defined in the same loop as x_i. In order to allow predicates from inner loops, that are necessary to support multiple exit nodes, those predicates are converted to the outer loop level by means of η functions.

Example 1. *The example in Figure 1c features a γ node that either selects x_1 or x_2, depending on the value of the Boolean variable i_1. In Figure 2c, there are two μ nodes in the TGSA representation, that select x_0 and y_0 in the first iteration of B_1, and x_2 and y_2 in subsequent loop iterations. There is also one η node, that selects the value assigned to x_1 in the last iteration of B_1.*

For the construction of the TGSA form, every ϕ node is replaced by a μ, η or γ node, depending on the basic block it is defined in. Second, the decision diagrams for η and γ nodes are computed by inspecting the CFG. The algorithm to compute the decision diagrams operates on the acyclic forward CFG; it is based on the algorithm presented in [12], but visits every basic block only once, using a topological order traversal.

We first describe the algorithm for γ nodes. The basic blocks that are direct predecessors of the block that defines the γ instruction are associated with expressions that evaluate to the corresponding reaching definition. These expressions constitute the leaves of the final decision diagram. Starting from the direct predecessors, all basic blocks reachable in the forward CFG using the predecessor relation are visited in *reverse* topological order, up to the immediate dominator of the ϕ node. For each block visited, we inspect the the decision diagrams that are associated with its successors, that have been visited before. If the same decision diagram D is associated with all visited successors, the block is associated with D as well. Otherwise, a new diagram is build, with the block's decision as root node, and the decision diagrams associated with the successor blocks as children of the root node. Finally, the decision diagram associated with the immediate dominator of the γ node provides the expression to select matching reaching definitions.

Example 2. *The decision diagram for x_3 in Figure 1b is computed as follows. First, we inspect the ϕ instruction defined in B_4, and associate B_2 with x_1 and B_3 with x_2. B_1 is a predecessor of B_2 in the (forward) CFG, and thus visited next. As the expressions associated with B_1's successors B_2 and B_3*

are not equivalent, a new decision diagram for B_1 is created. Its root is labelled with p and the two successors of the root node are the leaf nodes associated with B_2 (labelled with x_1) and B_3 (labelled with x_2). Now we note that B_1 is the immediate dominator of B_4, and thus the decision diagram associated with B_1 determines which definition reaches x_3. This decision diagram is subsequently used to defined the γ node in the TGSA representation in Figure 1c.

For η nodes (loop decision diagrams), a similar algorithm is employed. In this case, the traversal starts from the loop's latch and from the exit blocks of the loop. The decision diagrams of exit blocks are leaves that represent loop exit, and the one associated with the latch is a leaf that represents another loop iteration. The rest of the construction works the same way as for γ decision diagrams, and proceeds until the loop header is reached. The last argument to η nodes in exit blocks of the loop is set to the decision diagram for the loop header.

2.1.3 Input-Data Dependency Graphs

In order to provide a rigorous definition of input-data dependencies, we first present an executable semantics of programs in TGSA form, defined in Table 1. The evaluation function $\mathcal{I}$ computes the value of a variable in a given loop context by backward substitution. The loop context C is a sequence of loop iteration counters, one for every loop the definition is enclosed in. The evaluation of ordinary functions and γ nodes is straight forward. For μ nodes, the variable defined outside the loop is computed in the first loop iteration, and the variable defined in the previous loop iteration is computed otherwise. For η nodes, the index of the last loop iteration (i.e., the least index such that the loop decision diagram evaluates to `false`) is computed. The η node then evaluates the value of its first argument in the last loop iteration.

Input-data dependencies are defined in terms of a directed graph, using the executable semantics in Table 1. In the graph, there is an edge from every variable to its direct dependencies. If there is a path from x to y in the input-data dependency graph, then x may depend on y, otherwise x is independent from y. For instructions $x_{n+1} = \gamma(x_1, \ldots, x_n; D[p_1, \ldots, p_n])$, x_{n+1} depends on each of the reaching definitions x_i (ϕ dependency), and on each of the predicates p_i (γ dependency). For every definition $x_2 = \eta(x_1, E[p_1, \ldots, p_n])$, x_2 depends on the predicates p_i that determine the number of loop iterations (η dependency) and obviously on x_1 as well (def-use dependency). For all other instructions, if x_i is referenced in the definition of the variable x_{n+1}, x_{n+1} depends on x_i (def-use dependency).

Example 3. *The input-data dependency analysis is demonstrated by means of a slightly larger example in Figure 3. The CFG in TGSA form is shown in Figure 3c, and includes two decision diagrams, one for the loop exit condition of the only loop, and one for the γ node that defines r_5. Figure 3b illustrates the dependency graph for the example. The final value r_5 depends on all three inputs, as expected, where as the values computed for the else branch (r_3 and r_4) only depend on B and C.*

Type	Definition	Semantics
instruction	$x_{n+1} = f(x_1, \ldots, x_n)$	$\mathcal{I}(x_{n+1}, C) = f(\mathcal{I}(x_1, C), \ldots, \mathcal{I}(x_n, C))$
γ node	$x_{n+1} = \gamma(x_1, \ldots, x_n; D[p_1, \ldots, p_m])$	$\mathcal{I}(x_{n+1}, C) = \begin{cases} \mathcal{I}(x_1, C) & \text{if } s = x_1 \\ \vdots & \\ \mathcal{I}(x_n, C) & \text{if } s = x_n \end{cases}$ where $s = D[\mathcal{I}(p_1, C), \ldots, \mathcal{I}(p_m, C)])$
μ node	$x_3 = \mu(x_1, x_2)$	$\mathcal{I}(x_3, C \cdot \langle c_n \rangle) = \begin{cases} \mathcal{I}(x_1, C) & \text{if } c_n = 0 \\ \mathcal{I}(x_2, C \cdot \langle c_n - 1 \rangle) & \text{if } c_n > 0 \end{cases}$
η Node	$x_2 = \eta(x_1; E[p_1, \ldots, p_m])$	$\mathcal{I}(x_2, C) = \mathcal{I}(x_1, C \cdot \langle \mathrm{LB}(C, 0) \rangle)$ $\mathrm{LB}(C, i) = \begin{cases} \mathrm{LB}(C, i+1) & \text{if } p(i) \\ i & \text{if } \neg p(i) \end{cases}$ where $p(i) = E[\mathcal{I}(p_1, C \cdot \langle i \rangle), \ldots, \mathcal{I}(p_m, C \cdot \langle i \rangle)]$

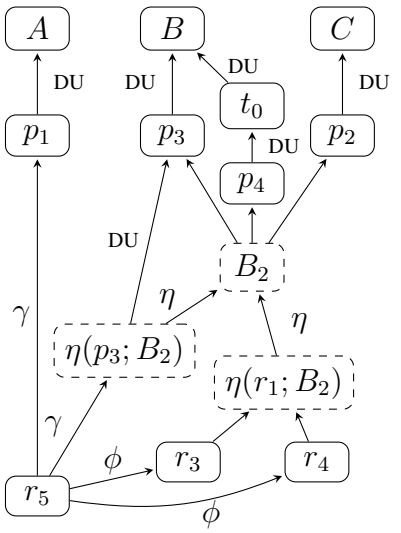
Table 1: Executable semantics for program representation in TGSA form

```
int f(int A, int B, int C) {
  int r;
  if(A == 0)
  {
    r = 1;
  }
  else
  {
    r = 0;
    while(1)
    {
      if(r > C)
      {
        if(r >= B ||
           2*r >= B+10)
          break;
      }
      r = r + 1;
    }
  }
  return r;
}
```

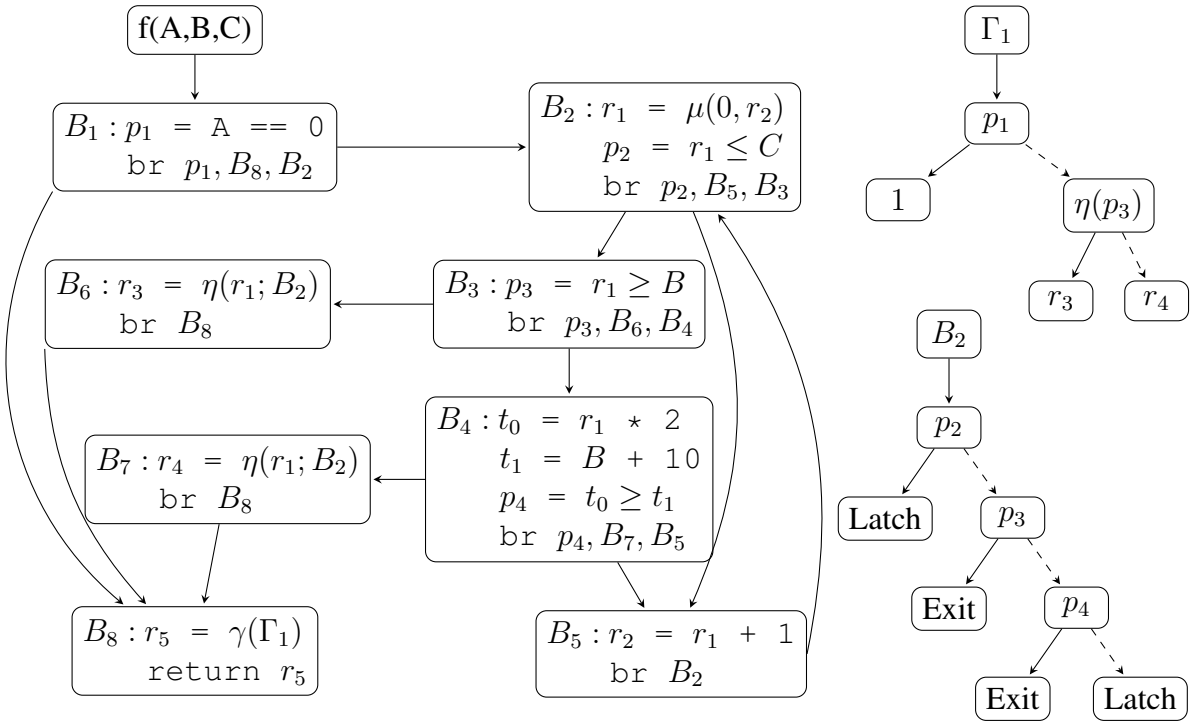
(a) Example Program Source

```
int f(int A, int B, int C) {
B1: p1 = A == 0
    br p1, B8, B2
B2: r1 = φ(B1 → 0, B5 → r2)
    p2 = r1 > C
    br B3, p2, B5
B3: p3 = r1 ≥ B
    br p3, B6, B4
B4: t0 = r1 * 2
    t1 = B + 10
    p4 = t0 ≥ t1
    br p4, B7, B5
B5: r2 = r1 + 1
    br B2
B6: r3 = φ(B3 → r1)
    br B8
B7: r4 = φ(B4 → r1)
    br B8
B8: r5 = φ(B1 → 1, B6 → r3,
           B7 → r4)
    return r5
}
```

(b) LCSSA Representation



(c) Input-Data Dependency Graph



(d) TGSA representation

Figure 3: This example illustrates our input-data dependency analysis. The TGSA graph includes decision DAGs that select the reaching definition at r_5 (Γ_1) and decide whether the loop is terminated (B_2). The input-data dependency graph illustrates that r_3 and r_4 depend on B and C, but not on A. For brevity, dependencies on r_1 and r_2 are not shown.

2.1.4 Global Memory and Function Calls

In order to implement a full-featured dependency analysis, it is necessary to deal with function calls, and with programs that use global memory, discussed next. Global memory is read using `load` instructions, and modified using `store` instructions. Global variables are accessed and modified using `load` and `store` operations as well, as usual in low-level program representations. The result of a `load` instruction depends on the dereferenced pointer, as well as on the contents of the possibly accessed memory locations. `store` instructions modify one out of several possible memory locations; the modification depends both on the value of the pointer and the value that is written to memory.

One approach to deal with global memory is to extend SSA to keep track of changes to memory. Similar to scalar variables, the value of a variable stored in global memory depends on different inputs at different program points, and might depend on loop iteration counts and on branch conditions. In order to integrate global memory into SSA form, it is thus necessary to distinguish the memory state at different program points, and explicitly merge the state of memory at control-flow join points. An example of this approach is the *store dependency graph* introduced in [29], a gated program representation similar to TGSA that explicitly models requests and modifications to the store. However, in order to be useful for dependency analysis, the program representation also would need to distinguish which parts of memory are affected by a store operation. As the set of memory locations possibly referenced by a pointer is usually dynamic and context dependent, this is a difficult problem.

In our approach, we do not directly model memory modifications in TGSA form. Instead, we calculate the dependencies of memory locations in a separate pass, using a flow-insensitive dependency model, and only consider `load` instructions during the TGSA-based input-data dependency analysis. The dependencies of a memory location are computed by first identifying all `store` instructions that might modify the memory location, using a standard alias analysis. In addition to both arguments of all `store` instructions that might modify the memory location, the value stored in the memory location also depends on all control-flow decisions that influence which store instruction is executed. Therefore, the value of a memory location depends on all control dependencies of all store instructions that might modify it. Furthermore, memory locations that might be subject to volatile accesses, or might be modified in concurrent tasks, are always considered to be input-data dependent. Finally, before the input-data dependency analysis, the set of memory locations possibly accessed by `load` instructions is determined. In addition to the pointer that is dereferenced, `load` instructions depend on all memory locations possibly accessed.

For interprocedural analysis, instructions of the form x_{n+1} `= call(`f`,` $x_1, \ldots, x_n$`)` need to be supported. Here, f has function type, and x_1 through x_n are the functions arguments. Two extensions to the intra-procedural model are necessary: on the one hand, we need to calculate the dependencies of x_{n+1}, and on the other hand, the dependencies of a variable in a function f needs to account for all call instructions that might invoke f. In addition to the function f and the arguments x_1 to x_n, the result of a call instruction might depend on `load` instructions. This kind of dependency is taken into account by adding all possibly *accessed memory locations* `MemDeps`$(f, x_1, \ldots, x_n)$, that might influence the result of a call instruction to the set of dependencies of x_{n+1}. The interprocedural dependencies of a variable defined in a function f are obtained by identifying all call sites of f, and adding dependencies between the formal parameters of f and the actual ones at the call sites. As usual, context dependent analyses are realised by means of call strings.

$$
\begin{aligned}
\mathcal{A}[x_1 := \texttt{constant}] &\equiv & x_1 &= \texttt{IDI} \\
\mathcal{A}[x_1 := \texttt{input}] &\equiv & x_1 &= \texttt{ID} \\
\mathcal{A}[x_{n+1} := \texttt{primop}(x_1, \ldots, x_n)] &\equiv & x_{n+1} &= \bigsqcup_{1 \le i \le n} x_i \\
\mathcal{A}[x_3 := \mu(x_1, x_2)] &\equiv & x_3 &= x_1 \sqcup x_2 \\
\mathcal{A}[x_2 := \eta(x_1, E[p_1, \ldots, p_m])] &\equiv & x_2 &= x \sqcup \bigsqcup_{1 \le i \le m} p_m \\
\mathcal{A}[x_{n+1} := \gamma(x_1, \ldots, x_n, D[p_1, \ldots, p_m])] &\equiv & x_{n+1} &= \bigsqcup_{1 \le i \le n} x_n \sqcup \bigsqcup_{1 \le i \le m} p_m \\
\mathcal{A}[x_{n+1} := \texttt{call}(f, x_1, \ldots, x_n)] &\equiv & x_{n+1} &= f \sqcup \bigsqcup_{1 \le i \le n} x_i \sqcup \bigsqcup_{m_i \in \texttt{MemDeps}(f, x_1, \ldots, x_n)} m_i \\
\mathcal{A}[P_i] &\equiv & & \bigsqcup_{x_i \in \texttt{args}(P_i)} x_i \\
\mathcal{A}[x_2 := \texttt{load}(x_1)] &\equiv & x_2 &= x_1 \bigsqcup_{m_i \in \texttt{MemLocs}(x_1)} m_i
\end{aligned}
$$

Figure 4: Dataflow equations for input-data dependence analysis

2.1.5 Input-Data Dependence Analysis

In previous works, input-data dependence was defined in terms of dataflow equations. Although it is straightforward to find the equations corresponding to our input-data dependency model, we provide them here for completeness. The operator $\mathcal{A}$ maps each statement, formal argument and memory location to one dependency equation. Following [10], the domain of the analysis is the semi-lattice $\{\text{ID}, \text{IDI}\}$; that is, definitions are either classified as being input-data independent (IDI), or being possibly input-data dependent (ID). A variable depends on two or more other variables is input-data independent only if all variables it depends on are; we therefore define $\text{ID} \sqcup \text{IDI} = \text{IDI} \sqcup \text{ID} = \text{ID}$.

A variable x_i is known to be input-data independent if the solution to the data-flow equations listed in Figure 4 gives $x_i = IDI$. The last three equations deal with interprocedural analysis and global memory access. The expression $\texttt{args}(P_i)$ takes a formal argument P_i to a function $f(\ldots, P_i, \ldots)$, and evaluates to the set of all actual arguments x_i for each call site $x_{n+1} = f(\ldots, x_i, \ldots)$. The expression $\texttt{MemDeps}(f, x_1, \ldots, x_n)$ evaluates to all memory locations that might influence the result of a function call $\texttt{call}(f, x_1, \ldots, x_n)$. The expressions $\texttt{MemLocs}(x)$ evaluates to all memory locations the pointer x might point to. The equations for a memory location m are determined in the following way: if m might be accessed in a non-predictable way, it is equated with ID. Otherwise, it is the join of all data and control dependencies of all `store` operations that might modify m.

2.1.6 Related Approaches

The program dependence web [21] combines a gated SSA representation with the PDG representation, complementing the selection of reaching definitions by "switches" that control whether a statement is executed. This program dependence web permits demand-driven execution, and would thus also be a well-suited representation for input-data dependence analyses. A different extension of SSA is the value dependency graph [29], that takes storage (or any other side effect) into account, by introducing explicit ordering dependencies between statements with side effects. This approach would increase the precision of our analysis for global variables, but it is not clear how to deal with possibly aliasing pointers in the general case.

Input-data dependency analysis in the context of real-time systems has been studied in [10], where it is used to guide the single-path transformation. In this article, input-data dependence is formalised as a system of dataflow equations, that relate variables which are either data or control dependent, similar to the dependency relation induced by a PDG. This previous input-data dependency analysis classifies all definitions and conditions in a basic block as input-data dependent, if some input data controls whether the basic block might be executed. As argued before, this is not a sufficiently precise notion of input-data dependence, as even constants might be classified as input-data dependent.

A problem area that benefits from input-data analysis is security analysis, where one tries to prove that certain variables or decisions do not depend on user input, or that there is no information leakage [11]. One approach in this area that is closely related to ours is that of Scholz, Zhang and Cifuentes [24]. It builds on augmented SSA graphs, a program representation that is similar the TGSA form. The authors define input-data dependence in terms of properties of this SSA graphs, and use dataflow equations to determine whether a variable is input-data dependent. Their definition of input-data dependency, however, is different, and not founded by semantic arguments. The authors posit that it is sufficient to consider forward control-dependencies (corresponding to γ dependencies) and so-called loop-control dependencies (μ nodes depend on branches inside the loop that if one branch target is an exit block). This dependency definition is not comprehensive, however, as the example in Figure 3 demonstrates: B_2 is neither a forward-control dependency, nor is one of its successors outside the loop, and thus it is not considered to be a control dependency. The approximation based on the post-dominance property that is used in their implementation suffers from the same correctness problem.

2.2 Code Guidelines based on Input-Data Independence

The inability to predict the success of loop bound analyses is a major obstacle for the construction of systems that are independent of analysis annotations provided by human experts. While for arbitrary code, there is little hope to find all necessary flow facts automatically, the situation is different if the implementation has to follow certain rules. For example, for single-path code, all necessary flow facts including loop bounds can be obtained by simply executing the program and recording the execution trace. Because the maximum time necessary to execute a real-time task needs to be bounded, there are input-data independent bounds on the the maximum number of iterations for every of the task's loops. As a consequence, every real-time task can be transformed to single-path form [23].

However, the single-path policy is sometimes perceived to be too restrictive, as it does not allow any input-data dependent control flow at all. Moreover, we are interested in a property that ensures that single-path conversion is possible without additional loop bounds that need to be provided by the software developer. The two classes of programs that we are now going to introduce rely on a precise definition of input-data independence, and permit automated control-flow analysis just as the single-path policy does.

Definition 1. *The class of programs with* input-data independent loop counters *consists of those programs, where for every loop and any context, the number of loop iterations is input-data independent.*

According to the semantics of the TGSA representation, loop bounds are solely determined by the predicates that appear in the inner nodes of loop decision diagrams. Therefore, if all predicates that appear in loop decision diagrams are provably input-data independent, then the loop iteration count of any loop is input-data independent. Note that the above is a formal code policy that is less restrictive than single-path, as it does not ban the use of all input-data dependent control flow decisions. For example, it is permissible to use state machines, without the need to transform the corresponding dispatch table to non-branching code, as required by the single-path policy. The policy is still sufficiently restrictive to guarantee that a simple and efficient form of abstract execution [9] will find the flow facts necessary for bounding the execution time, as demonstrated in the next section.

A super class of these programs that enjoys similar properties, but is slightly harder to identify, is obtained by relaxing the requirement that all predicates in a loop decision diagram need to be input-data independent.

Definition 2. *The class of programs with* input-data independent loop bounds *consists of those programs, where all loops are still bounded after replacing all input-data dependent branch conditions with non-deterministic choice.*

The intuition behind this second coding policy is that early-exit branches, which might reduce the number of loop iterations for some input, are not crucial for determining a loop bound. In order to identify such programs, consider the loop decision diagram $E[p_1, \ldots, p_n]$ associated with a loop, that evaluates to `true` in the last iteration and to `false` in previous iterations of the loop. We construct a new loop decision diagram $E'[p_{\sigma_1}, \ldots, p_{\sigma_k}]$, where p_{σ_i} is input-data independent, by universally quantifying all input-data dependent predicates. If E' is satisfiable, and it can be shown that in any context it evaluates to `true` eventually, the loop has an input-data independent iteration bound.

```
int bsearch_dep(int arr[],
  int N, int key)
{
  int lb = 0;
  int ub = N - 1;
  while (lb <= ub)
  {
    /* unsigned shift */
    int m = (lb + ub) >>> 1;
    if (arr[m] < key)
      lb = m+1;
    else if (arr[m] > key)
      ub = m-1;
    else
      return m;
  }
  return -1;
}
```

(a) Dependent loop bound

```
int bsearch_ilb(int arr[],
  int N, int key)
{
  int base = 0;

  for (int lim = N;
       lim > 0; lim >>= 1)
  {
    int p =
      base + (lim >> 1);
    if (key > arr[p])
      base = p + (lim&1);
    else if (key == arr[p])
      return p;
  }

  return -1;
}
```

(b) Independent loop bound

```
int bsearch_ilc(int arr[],
  int N, int key)
{
  int base = 0;
  int r = -1;

  for (int lim = N;
       lim > 0; lim >>= 1)
  {
    int p =
      base + (lim >> 1);
    if (key > arr[p])
      base = p + (lim&1)
    else if (key == arr[p])
      r = p;
  }
  return r;
}
```

(c) Independent loop counter

Figure 5: Input-Data Dependence in different Binary Search variants

Example 4. *As an example, compare the implementations of binary search given in Figure 5, assuming that the size of the array is input-data independent. All three implementations have a similar worst-case performance, but different characteristics when it comes to loop bound analysis. The first implementation (Figure 5a) requires a relatively complicated proof to establish the loop bound. In contrast, we know that it is always possible to calculate the precise loop bound for the implementation in Figure 5c, as the number of times the only loop is iterated is input-data independent. The implementation in Figure 5b is faster on average, as it uses an early-exit condition to leave the loop if the key was found. This program has an input-data independent loop bound, that can be obtained by abstracting the early-exit test and using the same technique as for the implementation on the right.*

It is of utmost importance that the developer of the system can ensure that code adheres to a given coding policy, in a systematic and modular way. To this end, the concept of input-data independence needs to be taken into account when defining the interface of functions or modules. On the one hand, the interface specification of a function should specify which parameters (and global variables) need to be input-data independent. On the other hand, the caller of the function has to ensure that those parameters which need to be input-data independent indeed are, possibly propagating dependencies to its own input-data dependence specification. In the binary search example, the caller is obliged to ensure that `N` is input-data independent. Using this assumption, the provider of the binary search function can ensure locally that loop bounds are input-data independent.

Global variables tend to introduce global dependencies, and thus it is advisable to avoid control-flow dependencies on globals. If however there is at least one use that requires the data stored in the variable to be input-data independent, modifications of this variable need to be restricted in all parts of the program. In particular, to ensure that the content of a global variable is input-data independent, it should not be modified by code that has input-data dependent control dependencies. From a development perspective, it seems advisable to decide early which global variables need to be input-data independent, and introduce automated checks that ensure that they indeed are.

Finally, we note that programs that have input-data independent loop counts or loop bounds can be converted single-path code in an automated way. Moreover, if loop bounds for every loop are provided in the form of external annotations, every program can be transformed into one with input-data independent loop bounds, using the loop conversion algorithm of [23]. Assume that a bound n, that is input-data independent but not necessarily constant, is known for some loop. The conversion algorithm introduces an additional loop counter, which is incremented by one in every loop iteration, and ensures the loop is left after n iterations. As the value of this loop counter is clearly input-data independent, a task will adhere to the policy if all problematic loops have been transformed in this way.

2.2.1 Efficient Flow-Fact Calculation

In this section, we will demonstrate an efficient way to derive all necessary flow facts for tasks with input-data independent loop bounds. Recall that for single-path code, all flow facts can be derived automatically by executing the task, and recording the instruction trace. As there is only one trace, counting the number of times a basic block is executed provides exact, absolute execution frequency counts.

The basic idea is similar for code with input-data independent loop bounds. However, we need to account for input-data dependent branch conditions, which result in more than one possible execution path. Instead of exact, absolute execution frequencies, we need to obtain bounds on loop iteration counts and on relative execution frequencies. The framework of abstract execution [9] provides all necessary techniques for this analysis.

However, in our setting abstract execution is vastly simplified, as we do not deal with input-data dependent variables at all. This allows to either preprocess the code eliminating all input-data dependent variables, or equivalently, us a flat abstract domain where an abstract value is either a concrete one or $\top$. Due to the removal of all input-data dependent assignments, it is not necessary to merge the values of ordinary program variables at any point. Only loop counters used to extract relative loop bounds need to be merged. This observation significantly reduces the complexity of the analysis.

Example 5. *In Figure 6a shows the second binary search implementation from Figure 5c, with all input-data dependent variables removed. The branches whose condition was input-data dependent have been replaced by non deterministic branches, as the condition variable is no longer available. In Figure 6b the instrumented program that is analysed to obtain the (context-dependent) loop bound is shown.*

Note that although we used the same programming language for the analysed input and the generated program carrying out abstract execution in this illustration, in practice we analyse the program at a lower representation level that is closer to the final executable. We believe that this form of abstract execution will not have any scalability issues in practice, though experiments with large programs have not been performed yet.

```
int bsearch_ilb(int _[], int N, int _)
{
  int lim;

  lim = N;
  while (lim > 0)
  {
    if (?)
      ;
    else if (?)
      return ?;
    lim >>= 1;
  }
  return ?;
}
```

(a) Abstracted Binary Search

```
int bsearch_ilb(int _[], int N, int _)
{
  int lim, _counter;

  lim = N;
  _counter = 0;
loop1:
  while (lim > 0)
  {
    _counter++;
    if (?) {
      record(CONTEXT, loop1, _counter);
      return ?;
    }
    lim >>= 1;
  }
  record(CONTEXT, loop1, _counter);
  return ?;
}
```

(b) Instrumented non-deterministic program

Figure 6: Control-flow analysis of input-data independent parts of the program

2.2.2 Discussion and Examples

The crucial question deciding the acceptance of the coding policies presented here is whether they are too restrictive for practical purposes. To this end, we argue that many algorithms we believe to be useful in typical hard real-time systems can indeed be designed to follow the policy in a natural way. Furthermore, it is possible to transform manually annotated loops into loops with an input-data independent loop bound. While this does not solve correctness and maintainability problems of annotations, it shows that all tasks can be written in a way adhering to the policy.

In the following, we discuss several important classes of algorithms in the context of input-data independent loop counters and loop bounds.

Digital Signal Processing Many algorithms used in digital signal processing do have natural single path implementations, given fixed array, matrix and block sizes. Examples include matrix multiplication, the discrete cosine transform (DCT), the discrete Fast Fourier Transform (FFT) and Finite Impulse Response (FIR) filters. The symbolic loop bound for FFT, for example, is not trivial to find. Given an input-data independent block size, however, the abstract execution technique suggested here determines precise loop bounds and relative execution frequencies, taking non-rectangular loop nests into account. For single-path algorithms, the complexity of calculating a loop bound is comparable to the complexity of simply executing the program.

Search and Sort Binary search has already served as an example to illustrate input-data independent loop counters (Listing 5). Insertion Sort and iterative variants of Merge Sort are sorting algorithms which use input-data independent loop counters if the size of the array to be sorted is input-data independent. This is not the case for Quick Sort, but this sorting algorithm is not suitable for hard real-time systems anyway, due to its poor worst-case performance.

State Machines State machines, which perform different actions depending on the current state, incur a high overhead when transformed to single-path code. This is because the actions of every state have to be carried out to conform to the single-path requirement. With the less restrictive policy presented here, if the loop counters in each action are input-data independent, state machines still may use dispatch tables.

Data structures with dynamic size The number of loop iterations of algorithms operating on data structures with a variable number of elements are usually bounded relative to the current size of the data structure, not relative to their maximal capacity. For these algorithms, it is necessary to use WCET-oriented variations of the algorithms, which are tailored towards the worst-case when the size reaches the data structures capacity. As it is necessary to distinguish undefined and defined entries, a certain overhead will be unavoidable here, and it still remains to be evaluated whether this is an acceptable strategy.

An interesting open question is whether implementations which adhere to the proposed policy are easier or more difficult to prove correct. As functional correctness is arguably even more important than analysability, it would be a strong case in favour of the policy if this was the case. While we do not know an answer in general, we note that the ability to detect loop bounds by means of static analysis is also beneficial for other program analysis tools. In particular, bounded model checkers, which are used to prove the absence of certain runtime errors (for example null pointer dereference or out of bound array indices) need to know all loop iteration bounds.

3 Coding Policies for Analysability

Section 3.1 briefly introduces static timing analysis and discusses challenges static timing analysis has to face. Section 3.2 investigates existing coding guidelines for their prospects to aid software predictability and discusses further means to increase the predictability of embedded software systems.

3.1 Static Timing Analysis

Exact worst-case execution times (WCETs) are impossible or very hard to determine, even for the restricted class of real-time programs with their usual coding rules. Therefore, the available WCET analysers only produce WCET guarantees, which are safe and precise upper bounds on the execution times of tasks. The combined requirements for timing analysis methods are:

- *soundness* – ensuring the reliability of the guarantees,
- *efficiency* – making them feasible in industrial practice, and
- *precision* – increasing the chance to prove the satisfaction of the timing constraints.

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived.

Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are – in part – responsible for the gap between WCET guarantees and observed upper bounds and between BCET guarantees and observed lower bounds. How much is lost depends on the methods used for timing analysis and on system properties, such as the hardware architecture and the analysability of the software.

Despite the potential loss of precision caused by abstraction, static timing analysis methods are well established in the industrial process, as proven by the positive feedback from the automotive and the avionics industries. However, to be successful, static timing analysis has to face several challenges, being discussed in the subsequent Section 3.1.2.

3.1.1 Tools for Static Worst-Case Execution Time Analysis

Figure 7 shows the general structure of WCET analysers like aiT, see `http://www.absint.com/aiT` – this is the static WCET tool we are most experienced with. The input binary executable has to undergo several analysis phases, before a worst-case execution time bound can be given for a specific task[1]. First, the binary is decoded (reconstruction of the control-flow). Next, loop and value analysis try to determine loop bounds and (abstract) contents of registers and memory cells. The (cache and) pipeline analysis computes lower and upper basic block execution time bounds. Finally,

[1] A task (usually) corresponds to a specific entry point of the analysed binary executable.

the path analysis computes the worst-case execution path through the analysed program (see [4] for a more detailed explanation).

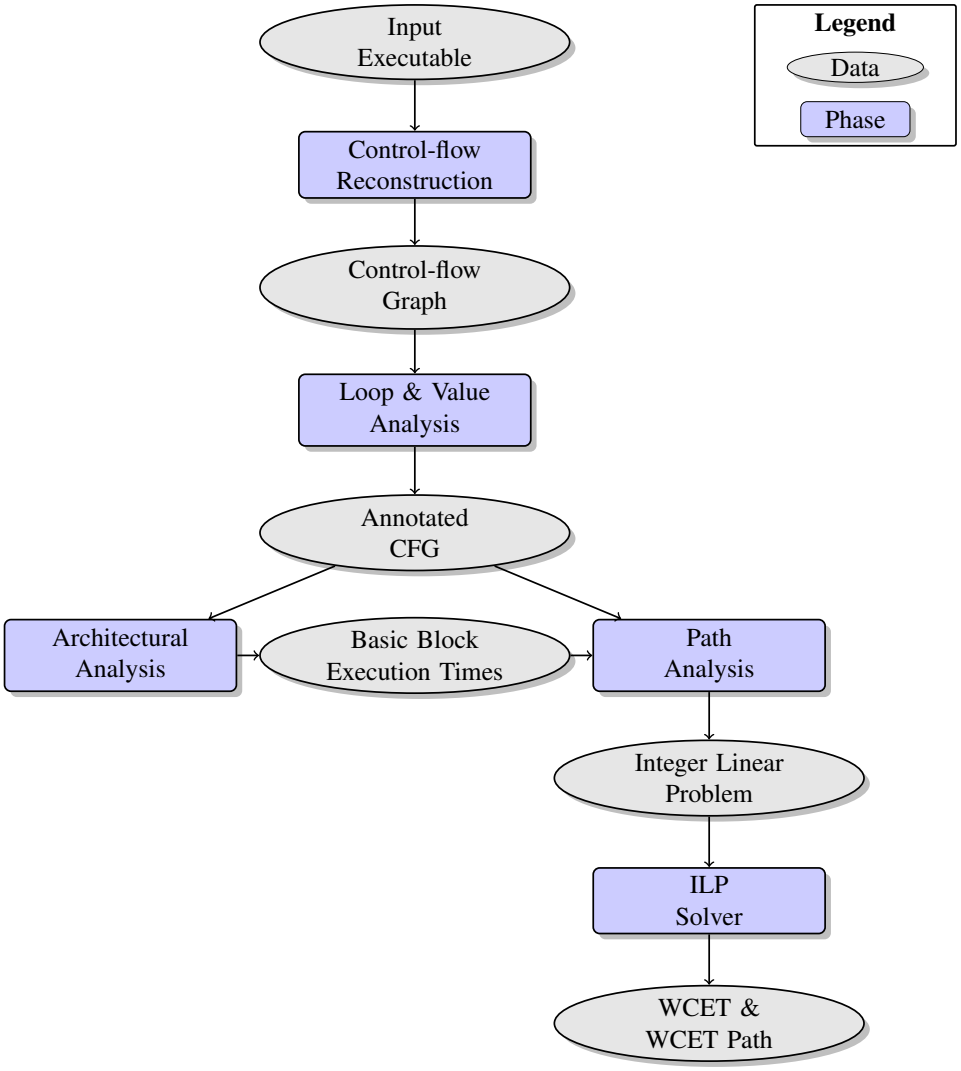


Figure 7: Components of a timing analysis framework and their interaction

3.1.2 Challenges

A static WCET analysis has to cope with several challenges to be successful. Basically, we discern two different classes of challenges. Challenges that need to be met to make the WCET bound computation feasible at all are *tier-one* challenges. *Tier-two* challenges are concerned with keeping the WCET bounds as tight as possible, e.g., to enable a feasible schedule of the overall system.

The used coding style is tightly coupled to the encountered tier-one challenges. Section 3.2 investigates whether coding guidelines that are in production use (indirectly) address such challenges and whether they ease their handling. Section 3.2.3 provides means how to cope with tier-two challenges.

In the following, we discuss tier-one WCET analysis challenges.

Function Pointers. Often simple language constructs do not suffice to implement a certain program behaviour. For instance, user-defined event handlers are usually implemented via function pointers to exchange data between communication library (e.g., for CAN devices) and the application. Resolving function pointers automatically is not easily done and sometimes not feasible at all. Nevertheless, function pointers need to be resolved to enable the reconstruction of a valid control-flow graph and the computation of a WCET bound.

Loops and Recursions. Loops (and also recursions) are a standard concept in software development. The main challenge is to automatically bound the maximum possible number of loop iterations, which is mandatory to compute a WCET bound at all. Whereas often-used counter loops can be easily bounded, it is generally infeasible to bound input-data dependent loops without additional knowledge. Similarly, such knowledge is required for recursions.

Irreducible Loops. Usually, loops have a single entry point and thus a single loop header. However, more complicated loops are occasionally encountered. By using language constructs like the `goto` statement from C or by means of hand-written assembly code, it is possible to construct loops featuring multiple entry points. So far, there exists no feasible approach to automatically bound this kind of loops [17]. Hence, additional knowledge about the control-flow behaviour of such loops is always required.

3.2 Software Predictability

In this section we discuss existing coding standards and investigate rules from the 2004 MISRA-C standard that are beneficial for software predictability. Thereafter, we describe how design-level information can further aid static timing analysis.

3.2.1 Coding Guidelines

Several coding guidelines have emerged to guide software programmers to develop code that conforms to safety-critical software principles. The main goal is to produce code that does not contain errors leading to critical failure and thus causing harm to individuals or to equipment. Furthermore, software development rules aim at improved reliability, portability, and maintainability.

In 1998, the Motor Industry Software Reliability Association (MISRA) published MISRA-C [19]. The guidelines were intended for embedded automotive systems implemented in the C programming language. An updated version of the MISRA-C coding guidelines has been released in 2004 [20]. This standard is now widely accepted in other safety-critical domains, such as avionics or defence systems. On the basis of the 2004 MISRA-C standard, the Lockheed Martin Corporation has published coding guidelines that are obligatory for Air Vehicle C++ development in 2005 [3]. Albeit certain rules tackle code complexity, there are no rules that explicitly aim at developing better time predictable software.

3.2.2 MISRA-C

Wenzel et al. [30] reckon that among the standards DO-178B, MISRA-C, and ARINC 653, only MISRA-C includes coding rules that can effect software predictability. In the following, we thus take a closer look at the 2004 MISRA-C guidelines. The list partially corresponds to the one found in [30] (focusing on 1998 MISRA-C), but refers to the potential impact on the time predictability using binary-level static WCET analysis (e.g., with the aiT tool).

Rule 13.4 (required): ***The controlling expression of a for statement shall not contain any objects of floating type.*** State-of-the-art abstract interpretation based loop analysers work well with integer arithmetic, but do not cope with floating point values [7, 5]. Thus, by forbidding floating point based loop conditions, a loop analysis is enabled to automatically detect loop bounds.

Rule 13.6 (required): ***Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.*** This rule promotes the use of (simple) counter-based loops and prohibits the implementation of a complex update logic of the loop counter. This allows for a less complicated loop bound detection.

Rule 14.1 (required): ***There shall be no unreachable code.*** Tools like aiT can detect that some part of the code is not reachable. However, static timing analysis computes an over-approximation of the possible control-flow. By this, the analysis might assume some execution paths that are not feasible in the actual execution of the software. Hence, the removal of unreachable parts from the code base leads to less sources of such imprecision.

Rule 14.4 (required): ***The goto statement shall not be used.*** The usage of the `goto` statement does not necessarily cause problems for binary-level timing analysis. These statements are compiled into unconditional branch instructions, which are no challenge to such analyses by themselves. However, the usage of the `goto` statement might possibly introduce irreducible loops into the program binary. There is no known approach available to automatically determine loop bounds for this kind of loops. Consequently, manual annotations are always required. Even worse, certain precision-enhancing analysis techniques, such as virtual loop unrolling [26], are not applicable.

Rule 14.5 (required): ***The continue statement shall not be used.*** Wenzel et al. [30] state that not adhering to this rule could lead to unstructured loops (see rule 14.4). However, `continue` statements only introduce additional back edges to the loop header and therefore cannot lead to irreducible loops. Any loop containing `continue` statements can be transformed into a semantically equivalent loop by means of `if-then-else` constructs. Hence, the only purpose of this rule is to enforce a certain coding style.

Rule 16.1 (required): ***Functions shall not be defined with a variable number of arguments.*** Functions with variable argument lists inherently lead to data dependent loops iterating over the argument list. Such loops are hard to bound automatically.

Rule 16.2 (required): ***Functions shall not call themselves, either directly or indirectly.*** Similarly to using `goto` statements, the use of recursive function calls might lead to irreducible loops in the call graph. Thus, a similar impact on software predictability would apply as discussed above for `goto` statements (see rule 14.4).

Rule 20.4 (required): ***Dynamic heap memory allocation shall not be used.*** Dynamic memory allocation leads to statically unknown memory addresses. This will lead to an over-estimation in the presence of caches or multiple memory areas with different timings. Recent work tries to address this problem by means of cache-aware memory allocation [13].

Rule 20.7 (required): ***The setjmp macro and the longjmp function shall not be used.*** In accordance to the discussion of rule 14.4 and of rule 16.2, the usage of the `setjmp` and the `longjmp` macro would allow the construction of irreducible loops. Hence, similar time predictability problems would arise.

3.2.3 Design-Level Information

Coping with all tier-one challenges of WCET analysis (see Section 3.1.2) is usually not sufficient in industrial practice. Additional information that is available from the design-level phase is often required to allow a computation of significantly tighter worst-case execution time bounds. Here, we address the most relevant tier-two challenges.

Operating Modes. Many embedded control software systems have different operating modes. For example, a flight control system differentiates between flight and ground mode. Any such operating mode features different functional and therefore different timing behaviour. Unfortunately, the modes of behaviour are not well represented in the control software code. Although there is ongoing work to semi-automatically derive operating modes from the source code [16], we still propose to methodically document their behavioural impact.

Such documentation could include loop bounds or other kinds of annotations specific to the corresponding operating mode. At best developers should instantly document the relevant source code parts to avoid a later hassle of reconstructing this particular knowledge.

Complex Algorithms. Complex algorithms or state machines are often modelled with tools like MATLAB or SCADE. By means of code generators these models are then transferred into, e.g., C code. During this process, high-level information about the algorithm or the state machine update logic respectively are lost (e.g., complex loop bounds, path exclusions).

Wilhelm et al. [31] propose systematic methods to make model information available to the WCET analyser. The authors have successfully applied their approach and showed that tighter WCET bounds are achievable in this fashion.

Data-Dependent Algorithms. Computing tight worst-case execution time bounds is a challenging task for strongly data-dependent algorithms. This is mainly caused by two reasons. *On the one hand*, data-dependent loops are hardly bounded statically. However, for computing precise WCET bounds, it generally does not suffice to assume the maximal possible number of loop iterations for each execution context. *On the other hand*, a static analysis is often unable to exclude certain execution paths through the algorithm without further knowledge about the execution environment. The following example demonstrates this problem.

Message-based communication is usually implemented by means of fixed-size read and write buffers that are reserved for each scheduling cycle separately. During an interrupt handler the message data is either copied from or to memory – depending on the current scheduling cycle. Here, read and write operations can never occur in the same execution context of the message handler. Without further information both operations cannot be excluded by a static WCET analysis. Additionally, the analysis has no a-priori information about the amount of data being transferred. However, the allocation of the data buffers and the amount of data to transmit is statically known during the software design phase. Using this information would allow for a much more precise static timing analysis of such algorithms.

Imprecise Memory Accesses. Unknown or imprecise memory access addresses are one of the main challenges of static timing analysis for two reasons. *First*, they impair the precision of the value analysis. Any unknown read access introduces unknown values into the value analysis and therefore increases the possibly feasible control-flow paths and negatively influences the loop bound analysis. In addition, any write access to an unknown memory location destroys all known information about memory during the value analysis phase. *Second*, the pipeline analysis has to assume that any memory module might be the target of an unknown memory access – the slowest memory module will thus contribute the most to the overall WCET bound. For architectures featuring data caches, an imprecise memory access invalidates large parts of the abstract cache (or even the whole cache) and leads to an over-approximation of the possible cache misses on the WCET path. Such unknown memory accesses can result from the extensive use of pointers inside data structures with multiple levels of indirections.

A remedy to this could be to document the memory areas that might be accessed for each function separately, especially if slow memory modules could be accessed. For example, memory-mapped I/O regions that are used for CAN or FLEXRAY controllers usually are only accessed in the corresponding device driver routines. Thus, the analysis would only need to assume for those specific routines that imprecise or unknown memory accesses target these (slow) memory regions. For all other routines, the analysis would be allowed to assume that different, potentially faster memory modules are being accessed.

Error Handling. In embedded software systems, error handling and recovery is a very complex procedure. In the event of an error, great care needs to be taken to ensure safety for individuals and machinery respectively.

A precise (static) timing analysis of error handling routines requires a lot more than the maximum number of possible errors that can occur or have to be handled at once. First of all however, it needs

Iteration Counts	Frequency of Occurrence	Observed for
0	1 552	
1	99 881 801	
2	116 421	
3	114	
4 .. 9	13	
10 .. 19	19	
20 .. 39	24	
40 .. 59	22	
60 .. 79	13	
80 .. 99	11	
100 .. 135	7	
156	1	`lDivMod(0xffd93580, 0x107d228)`
186	1	`lDivMod(0xfff2c009, 0x118dcc4)`
204	1	`lDivMod(0xffe870e3, 0x1414167)`

Table 2: Observed iteration counts for `lDivMod`.

to be decided whether the error case is relevant for the worst-case behaviour or not. If not, all error-case related execution paths through the software may be ignored during WCET analysis, which will obviously lead to much lower WCET bounds being computed. This however requires precise knowledge about which parts of the software are concerned with handling errors.

Otherwise, static timing analysis has to cope with error handling. The assumption that all errors might occur at once naturally leads to safe timing guarantees. However, in reality this is a rather uncommon or simply infeasible behaviour of the embedded system. Here, computing tight WCET bounds requires precise knowledge about all potential error scenarios. An early documentation of the system's error handling behaviour is thus expected to allow for a quicker and more precise analysis of the overall system.

Software Arithmetic. Under certain circumstances, an embedded software system makes use of software arithmetic. This is the case if the underlying hardware platform does not support the required arithmetic capabilities. For instance, the Freescale MPC5554 processor only supports single precision floating point computations [25]. If higher-precision FPU operations are required, (low-level) software algorithms emulating the required arithmetic precision come into play. Such algorithms are usually designed to provide good average-case performance, but are not implemented with good WCET predictability in mind. This often causes a static timing analysis to assume the worst-case path through such routines for most execution contexts.

An extreme example for a function with good average-case performance and bad WCET predictability is the library function `lDivMod` of the CodeWarrior V4.6 compiler for Freescale HCS12X. The purpose of this routine is to compute quotient and remainder of two 32 bit unsigned integers. The algorithm performs an iteration computing successive approximations to the final result. To get an impression on the number of loop iterations, we performed an experiment in which `lDivMod` was applied to 10^8 random inputs. Table 2 shows which iteration counts were observed in this experi-

ment. The number of iterations is 1 in more than 99.8% and 0, 1, or 2 in more than 99.999% of the sample inputs. On the other hand, iteration counts of more than 150 could be observed for a few specific inputs. There seems to be no simple way to derive the number of iterations from given inputs (other than running the algorithm). The highest possible iteration count could not yet be determined by mathematical analysis. Even if it were known that 204 is the maximum, a worst-case execution time analysis had to assume that such a high iteration number occurs when the input values cannot be determined statically, leading to a big over-estimation of the actual WCET.

To tighten the computed WCET bounds, further information would be required to avoid the cases with high numbers of loop iterations in many or all execution contexts. Making sure that the used software arithmetic library features good WCET analysability also helps to tighten the computed WCET bounds. Another – more radical – approach would be to employ a different hardware architecture that supports the required arithmetic precision.

3.3 Related Work

The impact of the source code structure on time predictability has been subject to several research papers and projects respectively.

Thiele and Wilhelm investigate threats to time predictability and propose design principles that support time predictability [27]. Among others the authors discuss the impact of software design on system predictability. For example, the use of dynamic data structures should be avoided, as these are hard to analyse statically.

Wenzel et al. [30] discuss the possible impact of existing software development guidelines (DO-178B, MISRA-C, and ARINC 653) on the WCET analysability of the software. Furthermore, the authors provide challenging code patterns, some of which, however, do not appear to cause problems for binary-level, static WCET analysis. For instance, calls to library functions do not necessarily impair the software's time predictability. The implementation and thus the binary code of the called function determines the time predictability, and not the fact of the function being part of a library. Nonetheless, the binary code of the library functions are required to be available to ensure a precise static worst-case execution time analysis if complex hardware architectures are being used. For ARINC 653 implementations that are truly modular this might not always be the case.

The purpose of the project COLA (Cache Optimisations for LEON Analyses) was to investigate how software can achieve maximum performance, whilst remaining analysable, testable, and predictable. COLA is a follow-on project to the studies PEAL and PEAL2 (Prototype Execution-time Analyser for LEON), which identified code layout and program execution patterns that result in cache risks, so called *cache killers*, and quantified their impact. Among others, the COLA project produced cache-aware coding rules that are specifically tailored to increase the time predictability of the LEON2 instruction cache.

The project MERASA aimed at the development of a predictable and (statically) analysable multi-core processor for hard real-time embedded systems. Bonenfant et al. [2] propose coding guidelines to improve the analysability of software executed on the MERASA platform. Both static analysis and measurement-based approaches are considered. In principle, these coding guidelines correspond to the MISRA-C guidelines discussed in Section 3.2.2.

4 Coding Policies in Industry

In practice, coding policies or coding guidelines are prevalent in industry. Typically, for safety-relevant software projects, policies based on a well-known standard like MISRA-2004 are devised and tailored to the context and specific needs of the project, individually.

The used coding policies typically contain measures to reduce the complexity of the software and hence improve their comprehensibility and testability. There might exist limits regarding the nesting depth and structure of control flow (e.g., single exit functions), restriction on the use of certain statements – most prominently `goto`, but in some cases also `break` and `continue` for loops – or the control flow graph complexity using the measure of *cyclomatic complexity* [18], the number of non-cyclic paths, or the call-graph complexity. Also, the use of global variables within functions might be limited and regulated, by demanding that they might only be accessed through accessor functions.

Safety-critical projects customarily employ tools for validation and verification of the software. Some tools require strict adherence to coding rules, as deviations are considered errors that must be fixed if they are not justified.

Experience has shown that certain code quality metrics are difficult to respect by the developers, especially those related to the size and complexity of the code (e.g., limits on number of statements within a function or restrictions on the cyclomatic complexity). There is a demand for a more automated development process, regarding software verification, documentation and requirements management.

We think that the presented coding policies for automated WCET analysis are easily employable and do not impose a considerable additional burden on the developers. The gain in analysability and hence tighter WCET bounds should outweigh the additional effort required.

5 Requirements

For the sake of completeness, we list the requirements from Deliverable D1.1 that target the compiler work package (WP5) and explain how they are met by the current version of the tool chain. NON-CORE and FAR requirements are not listed here.

5.1 Industrial Requirements

P-0-505 The platform shall provide means to implement preemption of running threads. These means shall allow an operating system to suspend a running thread immediately and make the related CPU available to another thread.

The compiler supports inline assembly, which can be used to implement storing and restoring threads. Further support will depend on the details of the implementation of preemption in the Patmos architecture, developed in the scope of interrupt virtualisation by our project partners (Task 2.6). There is no specific task devoted to the integration of preemption for TUV. Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory. We adapted the compiler and the C library to support compilation of the RTEMS operating system, which features context switching and POSIX threads.

P-0-506 The platform shall provide means to implement priority-preemptive scheduling (CPU-local, no migration).

The compiler supports inline assembly, which can be used to implement storing and restoring threads. Further support will depend on the details of the implementation of preemption in the Patmos architecture, developed in the scope of interrupt virtualisation by our project partners (Task 2.6). There is no specific task devoted to the integration of preemption for TUV. Though, we adapted the ISA to allow storing and restoring the contents of the stack cache to and from the external memory. We adapted the compiler and the C library to support compilation of the RTEMS operating system, which features context switching and POSIX threads.

C-0-513 The compiler shall provide means for different optimisation strategies that can be selected by the user, *e.g.*: instruction re-ordering, inlining, data flow optimisation, loop optimisation.

In the LLVM framework, optimisations are implemented as transformation passes. The LLVM framework provides options to individually enable each transformation pass, as well as options to select common optimisation levels which enable sets of transformation passes.

C-0-514 The compiler shall provide a front-end for C.

The `clang` compiler provides a front-end for C. The compiler has been adapted to provide support for language features such as variadic arguments and floating point operations on Patmos.

C-0-515 The compile chain shall provide a tool to define the memory layout.

The tool chain uses `gold` to link and relocate the executable. The `gold` tool supports linker scripts, which can be used to define the memory layout.

S-0-519 The platform shall contain language support libraries for the chosen language.

The `newlib` library has been adopted for the Patmos platform, which provides a standard ANSI C library.

A-0-521 The analysis tool shall allow defining assumptions, under which a lower bound can be found, *i.e.*a bound that is smaller than the strict upper bound, but still guaranteed to be $>= WCET$ as long as the assumptions are true (*e.g.*instructions in one path or data used in that path fit into the cache).

We adapted the `LLVM` compiler framework to automatically emit flow facts and value facts that are generated by the internal analyses performed by the compiler. The tool chain also supports generation of flow facts from execution traces. The compiler is able to emit debug information which is used by the aiT WCET analysis tool to retrieve source code level flow annotations. Flow annotations at binary level can be used to add additional flow constraints to the WCET analysis.

S-0-522 Platform and tool chain shall provide means that significantly reduce execution time (*e.g.*: cache, scratchpad, instruction reordering).

The LLVM framework provides several standard optimisations targeting execution time, such as inlining or loop unrolling. The data scratchpad memory can be accessed with dedicated macros, which allow the programmer to manually utilize this hardware feature. Instruction reordering is performed statically at compile-time to reduce the number of stall cycles in the processor. The stack cache provides a fast local memory to reduce the pressure on the data cache and to obtain a better WCET bound. The compiler features WCET analysis driven optimisations and optimisations utilising the Patmos hardware features that automatically further reduce the WCET bound.

P-0-528 The tool chain shall provide a scratchpad control interface (*e.g.*: annotations) that allows managing data in the scratchpad at design time.

The SPM API has been integrated into the tool chain, which contains both low-level and high-level functions that allow copying data between SPM and external memory and to use the SPM as buffer for predictable data processing, respectively. Accessing data items on the SPM is possible with dedicated macros that use the address space attribute, which is translated to memory access instructions with the proper type in the compiler backend.

C-0-530 The compiler may reorder instructions to optimise high-level code to reduce execution time.

Instructions are statically reordered to make use of delay slots and the second pipeline, and to minimise stalls during memory accesses.

C-0-531 The compiler shall allow for enabling and disabling optimisations (through *e.g.*: annotations or command line switches).

In the LLVM framework optimisations are implemented as transformation passes. The LLVM framework provides options to individually enable each transformation pass, as well as options to select common optimisation levels which enable sets of transformation passes.

C-0-539 The compiler shall provide mechanisms (*e.g.*: annotations) to mark data as cachable or uncachable.

Variables marked with the `_UNCACHED` macro are compiled using the the cache bypass instructions provided by Patmos to access main memory without using the data cache.

S-0-541 There shall be a user manual for the tool chain.

All tool chain source repositories contain a `README.patmos` file, which explains how to build and use the tools provided by the repository. Additional documentation of the tool chain can be found in the patmos-misc repository and in the Patmos handbook in the patmos repository. Further information about the LLVM compiler can be found in the LLVM user guide.[2]

5.2 Technology Requirements

C-2-013 The compiler shall emit the necessary control instructions for the manual control of the stack cache.

The compiler emits stack control instructions to control the stack cache, so that data can be allocated in the stack cache. The compiler employs optimisation to reduce the number of emitted stack control instructions.

C-4-017 The compiler shall be able to generate the different variants of load and store instructions according to their storage type used to hold the variable being accessed.

The compiler backend supports all variants of load and store instructions that are currently defined by the Patmos ISA at the time of writing. Support for annotations to select the memory type for memory accesses is provided by dedicated macros.

C-4-018 The storage type may be implemented by compiler-pragmas.

Support for annotations to select the memory type for memory accesses is provided by dedicated macros.

C-5-027 The compiler shall be able to compile C code.

The `clang` compiler provides a front-end for C. The compiler has been adapted to provide support for language features such as variadic arguments and floating point operations on Patmos.

C-5-028 The compiler shall be able to generate code that uses the special hardware features provided by Patmos, such as the stack cache and the dual-issue pipeline.

The compiler uses special optimisations to generate code that uses the method cache, the stack cache and the dual-issue pipeline.

C-5-029 The compiler shall be able to generate code that uses only a subset of the hardware features provided by Patmos.

All code generation passes that optimise code for the Patmos architecture, such as stack cache allocation and function splitting for the method cache, provide options to disable the optimisations and thus emit code that does not use the special hardware features of Patmos.

C-5-030 The compiler shall support adding data and control flow information (*i.e.*: flow facts) to the code, *e.g.*: in form of annotations.

[2] `http://llvm.org/docs/`

Our tool chain extracts flow information from the code automatically and provides the information to the WCET analysis. Furthermore, because the compiler emits debug information, the capabilities of `aiT` to process annotations on the source code can be utilised.

C-5-031 The compiler shall provide information about potential targets of indirect function calls and indirect branches to the static analysis tool.

The compiler emits internal information such as targets of indirect jumps for jump tables. The tool chain provides means to transform this information to the input format of the WCET analysis tool.

C-5-032 The compiler shall pass available flow facts to the static analysis tool.

The compiler emits internal information such as targets of indirect jumps for jump tables. Additional value facts and flow facts are emitted by the compiler based on information generated by internal analyses of the compiler. The tool chain provides means to transform this information to the input format of the WCET analysis tool.

6 Conclusion

We presented a precise definition of input-data dependencies, which is founded by semantic arguments. This definition allows us to define two useful classes of programs for hard real-time systems. For tasks with input-data independent loop bounds, it is possible to guarantee that all loop bounds will be detected by an efficient algorithm. Furthermore, it is feasible to carry out static checks that verify that the program adheres to the policy, and to construct tasks following this policy in a systematic way. We argued by means of examples that this policy, which originates from the single-path paradigm, is suitable for real-time systems.

Our experience with static timing analysis of embedded software systems shows that the analysis complexity varies greatly. As discussed above, the software structure strongly influences the analysability of the overall system. Existing coding guidelines, such as the MISRA-C standard, partially address tier-one challenges encountered during WCET analysis. However, solely adhering to these guidelines does not suffice to achieve worst-case execution time bounds with the best precision possible. We usually suggest to document the software system behaviour as early as possible – desirably during the software design phase – to tackle the tier-two WCET analysis challenges. Otherwise, achieving precise analysis results during the software development testing and validation phase might become a costly and time consuming process.

Acronyms

CFG Control-Flow Graph
DAG Directed Acyclic Graph
ILP Integer Linear Programming
IR Intermediate Representation
LCSSA Loop-Closed SSA
PDG Program Dependency Graph
SSA Static Single Assignment
SCC Strongly-Connected Component
SCR Strongly-Connected Region
TGSA Thinned Gated Single Assignment
WCET Worst-Case Execution Time

References

[1] Motor Industry Software Reliability Association et al. Misra-c 2004: Guidelines for the use of the c language in critical systems. *ISBN 0*, 9524156(2):3, 2004.

[2] Armelle Bonenfant, Ian Broster, ClÃ©ment Ballabriga, Guillem Bernat, Hugues CassÃ©, Michael Houston, Nicholas Merriam, Marianne de Michiel, Christine Rochange, and Pascal Sainrat. Coding guidelines for WCET analysis using measurement-based and static analysis techniques. Technical Report IRIT/RR–2010-8–FR, IRIT, UniversitÃ© Paul Sabatier, Toulouse, March 2010.

[3] Lookheed Martin Corporation. C++ coding standards for the system development and demonstration program, December 2005.

[4] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

[5] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[7] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[9] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Jan Gustafsson, Björn Lisper, Raimund Kirner, and Peter Puschner. Code analysis for temporal predictability. *Real-Time Syst.*, 32(3):253–277, 2006.

[11] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[12] Paul Havlak. Construction of thinned gated single-assignment form. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 477–499. Springer Berlin / Heidelberg, 1994.

[13] Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

[14] Gerard J Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[15] Benedikt Huber and Peter Puschner. A code policy guaranteeing fully automated path analysis. *10th International Workshop on Worst-Case Execution-Time Analysis*, Jul 2010.

[16] Philipp Lucas, Oleg Parshin, and Reinhard Wilhelm. Operating mode specific WCET analysis. In Charlotte Seidner, editor, *Proceedings of JRWRTC*, October 2009.

[17] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[18] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[19] The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in vehicle based software, 1998.

[20] The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems, October 2004.

[21] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[22] Peter Puschner and Alan Burns. Writing temporally predictable code. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Peter P. Puschner. Transforming execution-time boundable code into temporally predictable code. In *DIPES '02: Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems*, pages 163–172, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[24] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. Technical report, Mountain View, CA, USA, 2008.

[25] Freescale Semiconductor. e200z6 PowerPC Core Reference Manual, 2004.

[26] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2–3):157–179, 2000.

[27] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, 2004.

[28] Peng Tu and David Padua. Efficient building and placing of gating functions. In *ACM SIGPLAN Notices*, volume 30, pages 47–55. ACM, 1995.

[29] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–310, New York, NY, USA, 1994. ACM.

[30] Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of dependable software development guidelines on timing analysis. In *Proceedings of the 2005 IEEE Eurocon Conference*, pages 575–578, Belgrad, Serbia and Montenegro, 2005. IEEE Computer Society.

[31] Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, LNCS. Springer-Verlag, 2010. To appear.