A Code Policy Guaranteeing Fully Automated Path Analysis

Benedikt Huber[1] and Peter Puschner[1]

1 **Institute of Computer Engineering, Vienna University of Technology, Austria**
`{benedikt,peter}@vmars.tuwien.ac.at`

Abstract

Calculating the worst-case execution time (WCET) of real-time tasks is still a tedious job. Programmers are required to provide additional information on the program flow, analyzing subtle, context dependent loop bounds manually. In this paper, we propose to restrict written and generated code to the class of programs with input-data independent loop counters. The proposed policy builds on the ideas of single-path code, but only requires partial input-data independence. It is always possible to find precise loop bounds for these programs, using an efficient variant of abstract execution. The systematic construction of tasks following the policy is facilitated by embedding knowledge on input-data dependence in function interfaces and types. Several algorithms and benchmarks are analyzed to show that this restriction is indeed a good candidate for removing the need for manual annotations.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.77

1 Introduction

Worst-Case Execution Time (WCET) analysis is concerned with determining an upper bound for the time needed to execute a task or procedure on a particular architecture. It is a necessary prerequisite for verifying that a system meets its timing specification. A widely used approach for WCET analysis is to analyze the set of execution paths and the timing of instruction sequences separately. The latter also includes the analysis of hardware components with global state, such as instruction and data caches or pipelines. Finally, the results of both the high-level and low-level analysis are fed into a solver, which computes the maximum execution time [12].

Flow facts are constraints that describe restrictions on the set of possible execution sequences. The path analysis has to determine a set of finite execution sequences as an overapproximation to all possible execution paths. As each considered execution path has to be finite, it is necessary to bound loop iteration counts and recursion depths. Together with information about the target of indirect jumps, these bounds are sufficient and necessary to derive some WCET bound. Loop bounds are either detected using an automated loop bound analysis, or are specified by the programmer through annotations.

WCET analysis is primarily concerned with the execution time of machine code on one particular architecture. Therefore, knowledge about the execution paths of a task is only useful if it can be mapped to machine code in a reliable way. Due to compiler optimizations, source code annotations are difficult to map to machine code. Annotations on the assembler level are difficult and tedious to write, and cannot be reused when the program is recompiled.

One solution to this dilemma is to have both a language that allows to integrate, test and verify flow information, and compilers that are aware of these flow facts, and transform them into flow facts on the machine code level [7, 3]. While highly desirable, flow-fact aware compilers are still rare. More importantly, source code annotations tend to be error prone, and often stay untested.

Another potential solution to eliminate the need for machine code level annotations is to derive all necessary flow facts automatically using an automated loop bound analysis. A variety of dataflow techniques has been proposed and implemented (e.g. [8]), as well as more expensive methods such as symbolic model checking. The problem with these techniques is that it is hard to predict whether they will find all necessary flow facts. In the fields of compiler optimization and testing, where most techniques were developed, a successful analysis is useful, but not crucial for correctness. In the WCET analysis domain, however, failing to derive loop bounds requires the user to resort to manual annotations.

We think that being unable to predict the success of loop bound analysis is a major obstacle for building systems that do not depend on user annotations. 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. Single path code [9] is characterized by the fact that there are no input-data dependent decisions and consequently only one possible execution path. For single-path code, all flow facts can be obtained by simply executing the program and recording the execution trace. Moreover, it is possible to transform all programs with known loop bounds to single-path form [10]. However, the single path policy is perceived to be too restrictive, as it does not allow any input-data dependent control flow at all.

In this paper, we propose a formal code policy less restrictive than single path, which does not ban the use of all input-data dependent control flow decisions. It e.g. allows to use state machines, without the need to transform the corresponding dispatch table to non-branching code, as required by the single-path concept. The policy is still sufficiently restrictive to guarantee that a simple and efficient form of abstract execution [4] will find the flow facts necessary for bounding the execution time.

Intuitively, the policy presented in this paper requires that all loops have at least one exit condition independent of input data. We present a formal definition of input-data dependence, and automatically classify input-data independent decisions. To this end, we use a program analysis technique which is similar to the one given in [5], but provides a more liberal definition of input-data dependence. Furthermore, the notion of input-data independence can be included in the application programming interface (API), facilitating a systematic construction of programs known to be analyzable.

As an example, compare the two implementations of binary search given in Listing 1a and Listing 1b, assuming that the size of the array is known. Both implementations have a similar performance, but different characteristics when it comes to loop bound analysis. The classic implementation in Listing 1a requires a relatively complicated proof to establish the loop bound. In contrast, it is easy to calculate the loop bound for the implementation in Listing 1b. If the size of the array is input-data independent, the second implementation agrees with the proposed policy. The single path implementation, which allows to calculate the exact number of iterations for a given array size, is shown in Listing 1c.

The crucial question deciding the acceptance of this methodology is whether it is too restrictive or not. 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 exit condition. 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.

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

(a) Dependent loop counter

```c
int bsearch_idi(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 counter

```c
int bsearch_sp(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)
    if (key == arr[p]) r    = p;
  }
  return r;
}
```

(c) Single path

Listing 1 Different Implementations of Binary Search

Outline

In Section 2, we discuss the notion of input-data dependence, and give a formal definition, which is easy to check both manually and automatically. Section 3 defines the class of tasks with input-data independent loop counters, and shows how to extend function interfaces and the type system to capture the intended input-data independence of variables. In Section 4, we propose a simple and efficient abstract execution framework to extract precise loop bounds. In Section 5, examples of algorithms and real-time benchmarks are evaluated with respect to the policy. Finally, Section 6 discusses future work and concludes the paper.

2 Input Data Independence

To analyze the WCET of a function, it is in general necessary to assume additional restrictions on the initial state of variables. For example, the execution time of a function performing a binary search depends at least on the size of the input array. The tasks scheduled in hard real-time systems, however, must always have an absolute WCET bound, which is provided as input to the scheduling algorithm. The policy presented in this paper requires certain variables to be input-data independent with respect to real-time tasks.

An expression is input-data independent if its value at a specific instruction of one execution trace does not depend on the environment. This includes dependencies on sensor values, timers and other values obtained from outside the system, as well as dependencies on other tasks or the runtime system, for example due to shared variables.

This would be the ideal notion of input-data independence, but it is intractable to check automatically. For example, if an arbitrary function $f(x)$ returns a constant independent of its parameter x, then $f(x)$ is input-data independent even if x is a value obtained from the environment. But deciding automatically whether the result of some arbitrary function has a dependency on one of its inputs is not possible in general. We do not want to be too restrictive either: An expression should not be unconditionally classified as being input-data dependent only because one decision on the path to the corresponding instruction was.

For these reasons, we define input data dependence in terms of dataflow equations, which can be easily checked by machines and are still comprehensible by humans.

There is already a close correspondence between data dependencies in static single assignment (SSA) form [1] and input data dependencies. In SSA form, the dependencies between the uses of a variable and its definition are made explicit by subscripting variables with an index reflecting their definition site, and adding dedicated statements for merging reaching definitions. Dependencies due to control flow decisions and due to the mutation of fields or array elements are not captured by SSA though.

In Figure 1, the source language for the analysis is defined. The statement $v :=$ `read` assigns v to a statically unknown value obtained from an interaction with the environment. The language includes support for reading and writing elements of arrays and fields of composite types. Variable definitions are in SSA form, i.e. each variable only appears once on the left-hand side of an assignment, and definitions reaching an use site are explicitly merged by so called ϕ functions.

We now define input data dependence in terms of an operator $\mathcal{A}$, which maps each statement to one or more dataflow equations. Following [5], the domain of the analysis is the semilattice $\{ID, IDI\}$. A variable v_i is input-data independent if there is a solution to the data-flow equations listed in Figure 2 with $v_i = IDI$, and input-data dependent otherwise. If a variable depends on two or more other variables, it is input-data independent only if all variables it depends on are. Therefore we define $ID \sqcap IDI = IDI \sqcap ID = ID$.

$v := c$	Assign v to an integer constant
$v :=$ `read`	Assign v to a value obtained from the environment
$v := v_1 \circ v_2$	Assign v to $v_1 \circ v_2$, with $\circ \in \{+, -, *, <, \leq, =, \texttt{AND}, \texttt{OR}, \texttt{XOR}\}$
$v := v_1[v_2]$	Assign v to the element at position v_2 of the array v_1
$v[v_1] := v_2$	Set element v_1 of the array v to v_2
$v := v_1.F$	v is assigned to the field F of of v_1
$v.F := v_1$	The field $v.F$ is assigned to v_1
$v := \phi(v_1, \ldots, v_n)$	v is the reaching definition from the set $\{v_1, \ldots, v_n\}$
`bz` v	Conditional branch, following the “true” edge if $v = 0$

Figure 1 The input language for input-data dependence analysis

The equations dealing with constants, environment interaction and binary operators are straightforward. Dependencies between control-flow *decisions* and ϕ functions are defined in terms of decision branches. A decision branch of $y := \phi(x_1, \ldots, x_n)$ is a conditional branch `bz` v_c, which has a direct influence which definition of x will reach the merge point defining y. If `bz` v_c has an influence which definition reaches $y := \phi(x_1, \ldots, x_n)$, y not only depends on $\{x_1, \ldots, x_n\}$, but also on the condition variable v_c.

Formally, `bz` v_c is a decision branch if there are two paths p_1 and p_2 starting at `bz` v_c and a variable x_i such that (a) p_1 and p_2 only have the first statement (`bz` v_c) in common (b) p_1 includes the assignment to x_i (c) p_2 does not include the assignment to x_i (d) the last statement of p_2 is the merge point $y := \phi(x_1, \ldots, x_n)$. To ensure this definition is correct in the presence of loops, we require all code to be in Loop-Closed SSA form. The set of all decision branches of $y := \phi(x_1, \ldots, x_n)$ is denoted by $\mathcal{D}(y)$. The resulting dependencies are reflected in the equations for *phi* functions in Figure 2.

Dependencies due to the mutation of array elements or fields usually require an alias analysis, determining which statements might influence which fields. A straight-forward type-based alias analysis [2] is not suitable for arrays, as arrays with the same element type may be used for both static and dynamic data. Store-based alias analyses distinguish data based on the memory address or the allocation site. As they track the set of all memory locations a variable may point to, the outcome of these analyses is difficult to predict for humans.

Therefore, a type attribute `IDI` is introduced, that distinguishes input-data independent and input-data dependent array types. This is similar to the `const` attribute in C. If v is declared to be input-data independent, we write $\textit{idtype}(v) = \mathit{IDI}$, otherwise $\textit{idtype}(v) = \mathit{ID}$. In an assignment $v = v_1[v_2]$, v is input data independent if v_1 and v_2 are input-data independent, and $\textit{idtype}(v_1) = \mathit{IDI}$. Assignments to arrays need to be type checked. It is required that in an assignment $v[v_1] = v_2$ both v_1 and v_2 are input-data independent if $\textit{idtype}(v) = \mathit{IDI}$.

For fields of composite types, we need a more fine grained distinction than for arrays, where either all or no elements are declared to be input-data independent. Consider e.g. a datatype for resizable vectors, consisting of three fields, one for the data in the vector, one for the vector's size and one for its maximum capacity. While some algorithms may require the maximum capacity to be input-data independent, clearly neither the size nor the internal data field have to be.

As it is usually possible to define different composite data types for different purposes, fields of a composite datatype are explicitly declared as being input data independent. In Figure 2, we write $\textit{idtype}(v.F) = \mathit{IDI}$ to denote that field F of variable v is declared to

$$\begin{array}{lcl}
\mathcal{A}[v := c] & \equiv & v = IDI \\
\mathcal{A}[v := \texttt{read}] & \equiv & v = ID \\
\mathcal{A}[v := v_1 \circ v_2] & \equiv & v = v_1 \sqcap v_2 \\
\mathcal{A}[v := v_1[v_2]] & \equiv & v = v_1 \sqcap v_2 \quad \text{if } \mathit{idtype}(v_1) = IDI \\
 & & v = ID \quad \text{otherwise} \\
\mathcal{A}[v := v_1.F] & \equiv & v = v_1 \quad \text{if } \mathit{idtype}(v_1.F) = IDI \\
 & & v = ID \quad \text{otherwise} \\
\mathcal{A}[v := \phi(v_1, \ldots, v_n)] & \equiv & v = \bigsqcap v_1, \ldots, v_n \quad \sqcap \quad \bigsqcap_{\texttt{bz}\ c_i \in \mathcal{D}(v)} c_i
\end{array}$$

Figure 2 Dataflow equations for input-data dependence analysis (without type checking)

be input-data independent. Similar to arrays, a type checker has to ensure that $v.F$ is not declared being input data independent if the right hand side of the assignment $v.F = v_1$ is not.

The example in Figure 3 illustrates the concepts presented in this section. It consists of two loops with an input-data independent loop counter, and a conditional branch, whose condition is input-data dependent. In this example, `bz` b_2 is the only decision branch, deciding whether r_2 or r_3 reaches $r_5 = \phi(r_2, r_3)$. Therefore, r_5 depends on the condition variable b_2, and is input-data dependent.

3 Input-data Independent Loop Counters

In this section, we will define the class of tasks with input-data independent loop counters. We restrict ourselves to reducible loops [6], i.e., loops with a unique entry node, called the loop header. A conditional branch within the loop is a decision branch for this loop, when there is one outgoing edge that exits the loop.

▶ **Definition 1.** A task has *input-data independent loop counters*, if each loop has at least one input-data independent decision branch, which will eventually terminate the loop. A conditional branch `bz` v is input-data independent if its condition variable is classified as IDI.

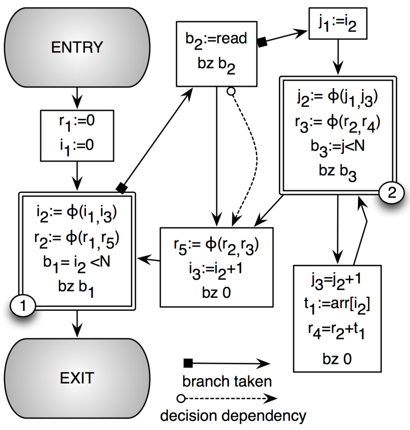
This definition captures those tasks whose loop iteration counts can still be determined after removing all input-data dependent variables. The example in Figure 3 has two loop decision branches, `bz` b_1 for the outer loop, and `bz` b_3 for the inner loop. As both are input-data independent, `dsum` indeed has input-data independent loop counters.

One important goal driving the definition above is that it should be possible to systematically construct tasks with input-data independent loop counters. To this end, the notion of input-data independence is included in the interface definition of functions. The interface specification of a function includes the set of parameters that need to be input-data independent. The caller of the function has to ensure that those parameters which need to be input-data independent indeed are, every time the function is called. In the example, the caller of `dsum` has to ensure N is input-data independent. Using this assumption, the analysis can prove that the function has input-data independent loop counters locally.

For composite data types and arrays, input-data independence is specified at the definition site. For example, the library provider specifies that the capacity field of a resizable vector always has to be input-data independent. The type system then has to check that no input-data dependent values are assigned to that field.

```
//@precondition: N = IDI
int dsum(int arr[], int N)
{
  int r=0;
  int i=0;
  while(i < N) {
    int c = read();
    if(c) {
      int j = i;
      while(j++ < N) {
        r+=arr[i];
      }
    }
    i=i+1;
  }
  return r;
}
```

Listing 1 Source Code for the Input-Data Dependence Analysis Example



(a) SSA Flow Graph

$$\begin{array}{lcl}
i_1 & = & IDI \\
i_2 & = & i_1 \sqcap i_3 \\
i_3 & = & i_2 \\ \hline
j_1 & = & i_2 \\
j_2 & = & j_1 \sqcap j_3 \\
j_3 & = & j_2 \\ \hline
N & = & IDI \\
t_1 & = & idtype(arr) \\ \hline
b_1 & = & i_2 \sqcap N \\
b_2 & = & ID \\
b_3 & = & j_2 \sqcap N \\ \hline
r_1 & = & IDI \\
r_2 & = & r_1 \sqcap r_5 \\
r_3 & = & r_2 \sqcap r_4 \\
r_4 & = & r_2 \sqcap t_1 \\
r_5 & = & (r_2 \sqcap r_3) \sqcap b_2
\end{array}$$

(b) DF Equations

$$\begin{array}{lcl}
i_1 & = & IDI \\
i_2 & = & IDI \\
i_3 & = & IDI \\ \hline
j_1 & = & IDI \\
j_2 & = & IDI \\
j_3 & = & IDI \\ \hline
N & = & IDI \\
t_1 & = & idtype(arr) \\ \hline
b_1 & = & IDI \\
b_2 & = & ID \\
b_3 & = & IDI \\ \hline
r_1 & = & IDI \\
r_2 & = & ID \\
r_3 & = & ID \\
r_4 & = & ID \\
r_5 & = & ID
\end{array}$$

(c) Solution

Figure 3 Example of the Input-Data Dependence Analysis

4 Generating Flow Facts by Abstract Execution

In this section, we will demonstrate an efficient way to derive all necessary flow facts for tasks with input-data independent loop counters.

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 counters. However, we need to take nondeterministic control flow branches into account. Instead of absolute execution frequencies, relative ones are recorded to obtain precise flow facts. The framework of abstract execution [4] provides all necessary notions for this analysis.

However, in our setting abstract execution is extremely simplified by eliminating all statements dealing with input-data dependent variables in a preprocessing step.

Generating bounds on relative loop iteration counts works by tracking and merging loop counters. A scope is the set of basic blocks associated with a method or loop. For each scope, loop counter and loop bound variables are introduced for every loop within the scope. The loop bounds are reset at the task entry. Loop counters are reset when a scope is entered. Each time the corresponding loop body is executed, the loop counter is increased. When the scope is left, the loop counter is read, updating the loop bound for the scope/loop pair. Additionally, one counter keeps track of the sum of innermost loops executed in a scope. In this way, it is possible to obtain precise flow facts when there are two or more inner loops with different dependencies on an outer loop counter.

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 the loop counters used to extract relative loop bounds need to be merged. This observation significantly reduces the complexity of the analysis.

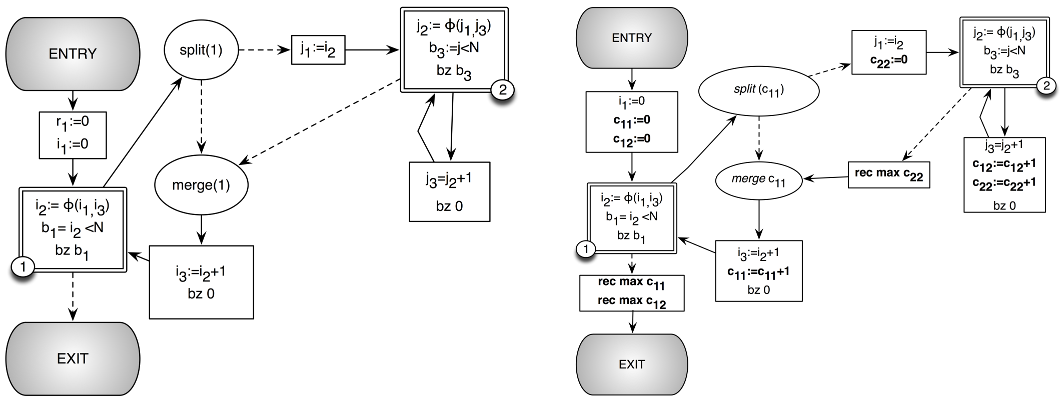
Figure 4a shows the simplified control flow graph of the `dsum` example from Figure 3, with all input-data dependent variables removed. Split points correspond to conditional branches, where the condition variable has been identified as input-data dependent. Note that control flow is split non-deterministically at these nodes, as the condition variable is no longer available.

In Figure 4b, the control flow graph for the abstract execution computing the flow facts for this example is shown. Note that although we used the same programming language for the analyzed input and the generated program carrying out abstract execution, this is not necessary in general. When only a low-level representation of the input is available, it might be desirable to use a higher-level language to generate code for abstract execution. Moreover, for languages with platform-dependent semantics, abstract execution has to faithfully interpret the characteristics of the target platform.

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.

5 Examples and Evaluation

This section discusses important classes of algorithms and tasks with input-data independent loop counters, and investigates the input-data independence of loop counters on a set of selected benchmarks.



(a) Reduced Flow Graph **(b)** Abstract Execution Flow Graph

Figure 4 Loop Bound Analysis of the `dsum` Example

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 e.g. FFT is not trivial to find. Given an input-data independent block size, the abstract execution technique from Section 4 determines precise loop bounds, 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 in the introduction (Listing 1). Insertion Sort and iterative 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. Quick Sort does not, but is unsuitable for hard real-time systems because of its poor worst-case performance.

State Machines

State machines, which perform different actions depending on the value of a state variable, incur a higher 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 our new code policy, the loop counters in each action are input-data independent, state machines need not be changed so that the task conforms to the policy.

Data structures with dynamic size

The loops of algorithms operating on data structures with a variable number of elements are usually bounded by a function based on the number of elements, not their maximal capacity. For these algorithms, different variants which are oriented towards the worst-case (size = capacity), need to be used. As it is necessary to distinguish undefined and defined entries, a certain overhead will be unavoidable here. It still remains to be evaluated whether this is an acceptable strategy.

Qualitative Benchmark Evaluation

We manually analyzed the properties of three applications available for the Java Optimized Processor (JOP) [11], and evaluated whether they conform with the policy. We found that most loop bounds do have input-data independent counters, while for those that do not, the dataflow analysis in JOP's WCET tool failed to derive loop bounds as well.

Lift benchmark: The Lift benchmark is the control loop of a simple lift controller. The task performs one out of a few different actions depending on its state and sensor values. All loops in the Lift benchmark have input-data independent counters, with most of them already being identified by the dataflow analysis integrated with JOP's WCET tool. When eliminating all input-data dependent assignments manually, 8 out of 13 methods are removed from the code.

Kfl Benchmark: The Kfl application is the software for a node in a distributed mast control application. All but two loops again had input-data independent counters. The remaining ones need to be annotated manually. The annotations are based on the programmer's knowledge that some global, static variable is always between 0 and 3. This is a non-obvious information, unlikely to be found by an automated analysis. This suggests to rewrite the offending code in order to avoid the annotation.

EjipCmp Benchmark: This benchmark is taken from an implementation of the UDP/IP stack used in a multi-core version of JOP. Some of its loops depend on the number of bytes a message contains. While there is a global limit to this bound, which depends on the size of the array used for storing the message, the message length is not input-data independent. In this case, the easiest way to conform to the policy is to add another exit condition based on the global limit.

6 Discussion

6.1 Source Code versus Machine Code

The ideas presented in this paper work on two different levels: The construction of predictable code applies to the source code level, while the flow fact generation for the compiled code applies to an optimized representation in some lower level language. This may either be low-level C, an internal representation of the compiler, or even machine code. For machine code, the flow graph reconstruction is not easy to automate though.

For the source code level, we need to provide a methodology for building or generating the code, and analysis tools to verify that the code meets the policy. The first goal is met by introducing annotations specifying input data independence via function interfaces and type annotations. To detect input-data dependencies, we perform a dataflow analysis on the source code. It is the responsibility of the programmer to ensure that input-data dependent branches terminate a loop eventually. If this is not the case, the program is considered to be faulty, and abstract execution may not terminate.

6.2 Functional Correctness vs. Timing Analysis

Proving the functional correctness is of course important too, so an interesting question is whether implementations which fulfill the proposed policy are easier or more difficult to prove correct. While we do not know an answer in general, 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 (null pointer dereference, out of bound array indices) need to know all loop iteration bounds.

7 Conclusion

In this paper, we have presented a formal definition for a code policy for hard real-time systems. For tasks with input-data independent loop counters, it is guaranteed that all loop bounds can be detected automatically. Furthermore, it is possible to check statically that the policy is fulfilled, and to systematically construct tasks following this policy. We have argued that this policy, which originates from the single-path paradigm, is suitable for real-time systems, and indeed characterizes a large set of analyzable code. Finally, a sketch of a static, efficient implementation of abstract execution to derive all loop bounds has been presented. We have recently started the implementation of the input-data dependency analysis. Future work includes implementation of the simplified abstract execution technique, removing statements dealing with input-data dependent prior to the analysis. Furthermore, we want to investigate suitable algorithms for dynamic data structures, and experiment with the analysis of machine code on ARM targets, which have served as a platform for single-path experiments in the past.

Acknowledgements

The authors would like to thank Wolfgang Puffitsch, Sven Bünte and the anonymous reviewers for their detailed and valuable comments. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design).

References

1 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.

2 Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117, New York, NY, USA, 1998. ACM.

3 Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-Aware C Compiler. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 121–126, Seoul/Korea, October 2006.

4 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.

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

6 Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

7 Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 29, Washington, DC, USA, 2001. IEEE Computer Society.

8 Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

9 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.

10 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.

11 Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

12 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.