Input-Dependency Analysis for Hard Real-Time Software

Jan Gustafsson and Björn Lisper
Department of Computer Science and Engineering,
Mälardalen University, Västerås, Sweden.
{jan.gustafsson,bjorn.lisper}@mdh.se
Raimund Kirner and Peter Puschner
Institut für Technische Informatik
Technische Universität Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

The execution time of software for hard real-time systems must be predictable. Further, safe and not overly pessimistic bounds for the worst-case execution time (WCET) must be computable. We conceived a programming strategy called WCET-oriented programming and a code transformation strategy, the single-path conversion, that aid programmers in producing code that meets these requirements. These strategies avoid respectively eliminate input-data dependencies in the code. The paper describes the formal analysis, based on abstract interpretation, that identifies input-data dependencies in the code and thus forms the basis for the strategies provided for hard real-time code development.

1 Introduction

One of the central demands on a real-time system is that it meets all timing requirements imposed by the application under guarantee. In order to give such a guarantee about the temporal correctness, engineers use specific design, implementation, and verification techniques. These techniques have to ensure the correct timing of entire applications down to the level of single tasks.

At the single-task level, a development framework has to aid the real-time programmer in a number of ways. Most essentially, the programmer needs to be provided with strategies for developing code with timing that is both predictable and easy to analyze. Further, techniques and tools for assessing the timing (e.g., WCET) of the code are needed. These tools need to provide the programmer with high-quality feedback about the temporal properties of the code.

We worked out a strategy for real-time task development that includes the above-mentioned services to the programmer: Tool support for *WCET-oriented programming* [Pus03] advises the programmer to write code that avoids algorithms or coding that produces code with a heavily input-data dependent control flow. Further, the *single-path conversion* [PB02, Pus02] converts the remaining or unavoidable input-data dependent alternatives in the control flow into code with a unique execution trace. The remaining code has a fully predictable timing and is easy to analyze for its WCET.

This paper describes techniques that can be used to implement tool support for WCET-oriented programming and the single-path conversion. Both WCET-oriented programming and single-path conversion rely on an effective analysis of input-data dependencies in control decisions in the code. We show how this analysis is realized by abstract interpretation and present the formal framework of the analysis in detail.

The paper is structured as follows: Section 2 gives a summary on WCET-oriented programming and single-path conversion. Section 3 introduces abstract interpretation and Section 4 describes the **While** language, a simple programming language that we use to define the semantics of our interpretation. Section 5 defines the abstract domain, states, and rules of the interpretation used for the input-dependency analysis. Section 6 illustrates our approach with an example, and Section 7 concludes the paper.

2 WCET-Oriented Programming and Single-Path Conversion

In previous work [Pus03] we argued that the algorithms and code structures used for hard real-time systems need to be very different from those algorithms and structures that are traditionally used. The basic argument against using traditional, non real-time code for hard real-time systems was that this code is written to achieve a good average-case performance and strongly relies on input-data dependent optimizations. We demonstrated, however, that the worst-case execution time of code with data-dependent control flow is typically higher than that of code that avoids data dependent control decisions. Based on this observation we introduced the concept of *WCET-oriented programming* [Pus03]:

> WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control flow decisions or, if this cannot be completely achieved, restricts the number of operations that are only executed for a subset of the input-data space to a minimum.

The restrictive use of input-data dependent alternatives as found in WCET-oriented code does not only keep the WCET down. It also keeps the total number of different execution paths through the code low. Identifying and operating on a smaller number of paths for WCET analysis is easier and therefore much less error-prone than dealing with a big number of alternatives. In this way, WCET-oriented programming does not only produce code with better WCETs but also yields more dependable WCET-analysis results than traditional programming.

As we mentioned above, WCET-oriented programming does not guarantee to produce code that is completely free of input-data dependent control decisions. In some cases, the semantics of a given problem or the limitations of the programming language used make it impossible to treat all input scenarios identically. As the remaining data dependencies may still impose a problem for WCET analysis, we conceived single-path programming [PB02] and the single-path conversion [Pus02] that removes input-data dependent control flows from the code.

The single-path conversion transforms the code for hard real-time systems into code with a unique execution trace. This is achieved by removing all input-data dependent branching operations from the code. To this end, both alternatives and loops with input-dependent control decisions are translated into deterministic code. Alternatives are converted by if-conversion [AKPW83]. Loops with input-data dependent control conditions are transformed by a more complex transformation that yields loops with constant iteration counts [Pus02].

3 Abstract Interpretation

Our aim is to calculate the run-time behavior of a program without having to run it on *all* input data, and while guaranteeing termination of the analysis.

One such technique for program analysis is *abstract interpretation* [CC77, Gus00], which means to calculate the program behavior using value descriptions or *abstract values* instead of real values. The price to be paid is loss of information; the calculation will sometimes give only approximate information. Abstract interpretation has three important properties:

1. It yields an *approximate* and *safe* description of the program behavior.
2. It is *automatic*, i.e., the program does not have to be annotated.
3. It works for *all* programs in the selected language.

It is important that the approximations for the concrete values are selected to *reduce the necessary calculations* in each step. But, in general, loss of precision is often the consequence of less calculation effort.

4 The While Language

For the sake of simplicity, we use the **While** language [NN92], with a syntax and semantics similar to common imperative languages like C and Pascal. We have added arrays to the language. This extension examplifies how the syntax and semantics of **While** can be extended with further constructs to form a complete language.

Syntax. The **While** language is built from the following syntactic sets:

- the set of statements, STATEMENT.
- the set of arithmetic expressions, AEXP;
- the set of boolean expressions, BEXP;
- the set of variables in the program, VARIABLE $= \{\texttt{a}, \ldots, \texttt{z}\}$;
- the set of numerals, NUM; and
- the truth values, $T = \{tt, ff\}$;

We use the following meta-variables when we describe the semantics of **While**:

- $var \in$ VARIABLE;
- $n \in$ NUM;
- $a \in$ AEXP;
- $b \in$ BEXP; and
- $S \in$ STATEMENT.

We use the following syntactical formation rules for programs written in **While** (where typewriter text represents source text, and italics represents meta-variables):

$$\begin{array}{rcl}
\textsc{Aexp} & ::= & n \mid var \mid var[a] \mid a_1{+}a_2 \mid a_1{-}a_2 \\
\textsc{Bexp} & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 < a_2 \mid \\
 & & a_1 <= a_2 \mid a_1 > a_2 \mid a_1 >= a_2 \mid \\
 & & \texttt{not}(b) \mid b_1 \& b_2 \\
\textsc{Statement} & ::= & var := a \mid var[a_1] := a_2 \mid \\
 & & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \\
 & & \texttt{while}\ b\ \texttt{do}\ S \mid S_1; S_2 \mid \texttt{skip}
\end{array}$$

State. The state σ is a mapping from variables to values (integers). Each element in an array is regarded as a separate variable. We use the notation $\sigma = [\texttt{x} \mapsto 0]$ to denote a state where x is assigned the value 0. Since σ is a mapping from variables to values, the expression $\sigma(\texttt{x})$ will give the result 0. With $\sigma_1 = \sigma[\texttt{x} \mapsto 1]$ we denote the updated state σ_1 where x is re-assigned to the value 1.

Semantics. We will use a structural operational semantics for **While** as defined in [NN92], Chapter 2.2, as shown in Figure 1. We have added rules for arrays.

- $\mathcal{N}$ defines the meaning of numerals $n \in$ Num;
- $\mathcal{A}$ the meaning of arithmetic expressions $a \in$ Aexp; and
- $\mathcal{B}$ the meaning of boolean expressions $b \in$ Bexp.

The meaning of a program (i.e., a sequence of statements) is defined as the final state that can be reached after a number of transitions. Each transition has the form $\langle S, \sigma\rangle \Rightarrow \gamma$ where γ is either an intermediate state $\langle S', \sigma'\rangle$ or a final state σ'.

The rules in Figure 2 define the transition $\Rightarrow$. The Assign rules update the state with the values of the right hand sides. The If rules select the true or false alternative, depending on the value of the condition. The While rule repeats the statement until the condition is false, when it terminates with a skip statement. The Seq(1) rule is used when the the first statement in a sequence goes to an intermediate statement and state (which is the case for the If, While and Sequence rules). The Seq(2) rule is used when the first statement terminates immediately (which is the case for the Assign and Skip rules). The Skip rule does nothing.

4.1 The Control Flow Graph and Data Flow Functions

Since the **While** language is deterministic, the transitions can be used to define functions from states to states. By $[\![S]\!]$ we denote the function that can be created from the transition of statement S. These functions can be visualized in control graph structures, see Figure 3. For compound statements, we get the structured constructs sequence, selection and iteration. Since the language is structured, it is possible to build any control flow structure possible in the language by recursively combining these structures.

Example. The simple example below shows the application of the semantics. The meaning of a statement is the final state after a sequence of transitions. We also show the flow graph of the example.

Statement:
$S =$ `while x>0 do` S',
where S'= `(a[2]:=a[x]; x:=x-1)`

Initial state:
$\sigma_1 = [\texttt{x}\mapsto 1, \texttt{a[1]}\mapsto 3, \texttt{a[2]}\mapsto 2]$

Transition sequence:

$\langle \texttt{while x>0 do (a[2]:=a[x]; x:=x-1)}, \sigma_1\rangle \Rightarrow$

$\langle \texttt{if x>0 then (}S'\texttt{; }S\texttt{) else skip}, \sigma_1\rangle \Rightarrow$

(since $\mathcal{B}[\![\texttt{x>0}]\!]\sigma_1 = tt$)

$\langle S'\texttt{; } S, s_1\rangle \Rightarrow$

(since $\mathcal{S}[\![\texttt{a[2]:=a[x]}]\!]\sigma_1 \Rightarrow \sigma_1[\texttt{a[2]} \mapsto 3]$ and $\mathcal{S}[\![\texttt{x:=x-1}]\!]\sigma_1[\texttt{a[2]}\mapsto 3] \Rightarrow \sigma_1[\texttt{a[2]}\mapsto 3, \texttt{x}\mapsto 0]$)

$\langle \texttt{while x>0 do (a[2]:=a[x]; x:=x-1)},$
$\sigma_1[\texttt{a[2]}\mapsto 3, \texttt{x}\mapsto 0]\rangle \Rightarrow$

$\langle \texttt{if x>0 then (}S'\texttt{; }S\texttt{) else skip},$
$\sigma_1[\texttt{a[2]}\mapsto 3, \texttt{x}\mapsto 0]\rangle \Rightarrow$

(since $\mathcal{B}[\![\texttt{x>0}]\!]\sigma_1 = ff$)

$\langle \texttt{skip}, \sigma_1[\texttt{a[1]}\mapsto 3, \texttt{x}\mapsto 0]\rangle \Rightarrow$

$\sigma_1[\texttt{a[2]}\mapsto 3, \texttt{x}\mapsto 0] =$

$[\texttt{x}\mapsto 0, \texttt{a[1]}\mapsto 3, \texttt{a[2]}\mapsto 3] = \sigma_2$

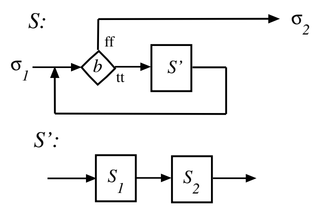
$S =$ `while` b `do` S'

$b =$ `x > 0`

$S' = S_1; S_2$

$S_1 =$ `a[2] := a[x]`

$S_2 =$ `x := x - 1`



$$\begin{aligned}
\mathcal{A}[\![n]\!]\sigma &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![var]\!]\sigma &= \sigma(var) \\
\mathcal{A}[\![var[a]]\!]\sigma &= \sigma(var[\mathcal{A}[\![a]\!]\sigma]) \\
\mathcal{A}[\![a_1{+}a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{A}[\![a_1{-}a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![\mathtt{true}]\!]\sigma = tt & \qquad \mathcal{B}[\![\mathtt{false}]\!]\sigma = \mathit{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma = \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 < a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma < \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 {<=} a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma \leq \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 > a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma > \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 {>=} a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma \geq \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![\mathtt{not}(b)]\!]\sigma &= \neg\mathcal{B}[\![b]\!]\sigma \\
\mathcal{B}[\![b_1 \& b_2]\!]\sigma &= \mathcal{B}[\![b_1]\!]\sigma \wedge \mathcal{B}[\![b_2]\!]\sigma
\end{aligned}$$

Figure 1. Semantics of expressions

Assign to variable:	$\langle var := a, \sigma\rangle \Rightarrow \sigma[var \mapsto \mathcal{A}[\![a]\!]\sigma]$
Assign to array:	$\langle var[a_1] := a_2, \sigma\rangle \Rightarrow \sigma[var[\mathcal{A}[\![a_1]\!]\sigma] \mapsto \mathcal{A}[\![a_2]\!]\sigma]$
If(t):	$\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \Rightarrow \langle S_1, \sigma\rangle \qquad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$
If(f):	$\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \Rightarrow \langle S_2, \sigma\rangle \qquad \text{if } \mathcal{B}[\![b]\!]\sigma = \mathit{ff}$
While:	$\langle \mathtt{while}\ b\ \mathtt{do}\ S, \sigma\rangle \Rightarrow \langle \mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ b\ \mathtt{do}\ S)\ \mathtt{else}\ \mathtt{skip}, \sigma\rangle$
Seq(1):	$\dfrac{\langle S_1, \sigma\rangle \Rightarrow \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \Rightarrow \langle S_1'; S_2, \sigma'\rangle}$
Seq(2):	$\dfrac{\langle S_1, \sigma\rangle \Rightarrow \sigma'}{\langle S_1; S_2, \sigma\rangle \Rightarrow \langle S_2, \sigma'\rangle}$
Skip:	$\langle \mathtt{skip}, \sigma\rangle \Rightarrow \sigma$

Figure 2. Transitions

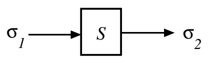
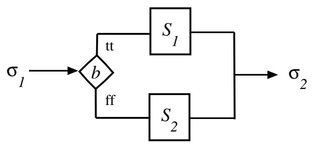
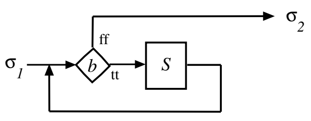
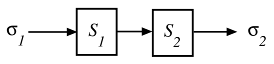
Statement	Data Flow Function	Control Flow Graph
S	$\sigma_2 = [\![S]\!]\sigma_1$	
`if` b `then` S_1 `else` S_2	$\sigma_2 = \text{if } B[\![b]\!]\sigma_1 \text{ then } [\![S_1]\!]\sigma_1 \text{ else } [\![S_2]\!]\sigma_1$	
`while` b `do` S	$\sigma_2 = \text{if } B[\![b]\!]\sigma_1 \text{ then} [\![\mathtt{while}\ b\ \mathtt{do}\ S]\!]([\![S]\!]\sigma_1) \text{ else } \sigma_1$	
$S_1; S_2$	$\sigma_2 = [\![S_1'; S_2]\!]([\![S_1]\!]\sigma_1)$ -- Seq(1) $\sigma_2 = [\![S_2]\!]([\![S_1]\!]\sigma_1)$ -- Seq(2)	

Figure 3. Data flow functions and control flow graphs for statements

5 Abstract Interpretation of While Programs to find Input Dependencies

The purpose of the program analysis is to find the properties of conditions in programs. A condition is defined to depend on input data if any of the variables included in the condition is input dependent. This means that the analysis first calculates input data dependencies for variables, then draws conclusions about the conditions. We differ between two cases:

1. The condition (e.g., in if, switch and loop-statements) is *not* depending on input data.
2. The condition *may* depend on input data.

For the first case, the code generation may generate ordinary code, while for the second case, single path conversion using predicated instructions [Pus02] has to be used to assure single path behavior.

5.1 Abstract Program

The analysis is a *semantics-based abstract interpretation*. We will, as a preparation to the abstract interpretation, transform a program to a corresponding abstract program. Interpretation of this abstract program, using abstract semantic rules and abstract values, will yield a final value for the variables in each program point, which is a safe approximation of the corresponding concrete values. “Safe” in this context means that all real executions are always “covered” by the abstract executions.

We can use this information to, in a safe way, determine whether ordinary or predicated instructions should be generated for the conditions in the program. For the two cases mentioned above, safety means that ordinary code is always correctly generated for input data dependent conditions. Sometimes, however, predicated instructions may be generated for non-input dependent conditions due to overestimations by the abstract interpretation.

5.2 Abstract Domain

All abstract variables in the program will be mapped to the abstract domain below This mapping is called the *abstract state* and is denoted with $\tilde{\sigma}$.

$$\begin{tikzcd}
\texttt{ID} \arrow[d, dash] \\
\texttt{NID}
\end{tikzcd}$$

The values in the Hasse diagram have the following explanation:

- *ID* marks a value that may be input dependent;
- *NID* marks a value that is not input dependent.

The order in the diagram represents the information content of the value. Also, the order defines the effect of the least upper bound ($\sqcup_{var}$) operation, used for disjunctions of variable values. For example, if a value is `ID` or `NID`, the result will be the safe (over)approximation $\texttt{ID} \sqcup_{var} \texttt{NID} = \texttt{ID}$.

The least upper bound for abstract states ($\sqcup$) is defined as $\sqcup_{var}$ for each variable. Formally

$$(\tilde{\sigma}_1 \sqcup \tilde{\sigma}_2)(var) = \tilde{\sigma}_1(var) \sqcup_{var} \tilde{\sigma}_2(var)$$

for all variables var in the resulting state.

5.3 Initial Abstract State of a Program

In the initial abstract state of a program, the abstraction function α will set all input data dependent variables (corresponding to e.g., input parameters to C functions) to *ID*, and the rest of the variables to *NID*.

Arrays are abstracted in a special way. For each concrete array (e.g. $\texttt{a}[i]$, $i = 1, \ldots, n$) there is *one* abstract variable `a[]` representing *all* values in `a`. The reason for this is simplicity - the simple abstract domain means that we cannot keep track of the indeces in arrays. At initialization, we first for each of its elements `a[i]` calculate the abstract value $\alpha(\texttt{a[i]})$ as for ordinary variables. Then we set `a[]` $= \sqcup_{i=1}^{n}(\alpha(\texttt{a}[i]))$.

We also add an extra variable to the state, `flow`, representing the data dependency of the control flow in the program at the current program point. We will set `flow` to *NID* at the beginning of the program. Actually, it will stay set to *NID* until the control flow is somehow controlled by an input data dependent variable.

5.4 Abstract Semantic Rules

The rules defines in Figure 5 the abstract transition $\tilde{\Rightarrow}$. The rules in the figure need some explanation.

- Assign to variable is similar to concrete assign.
- Assign to array assigns a value to `a[]` which is a safe overestimation of all possible values in the array.
- If(*NID*) describes the case when the value of the condition `b` is not input dependent (*NID*), Then we analyze both possible edges and form the least upper bound of the results.
- If(*ID*) is the most complex rule. If the value of the condition `b` may be input dependent (*ID*), we analyze both possible edges and form the least upper bound of the results. Note that `flow` is set to ID for both edges. When the analysis of the expression is finished, the value of `flow` is reset to its original value. The reason for this handling of `flow` is illustrated by the example in Section 6.
- The While, Seq and Skip rules are similar to the concrete case.

5.5 Abstract Data Flow Equations

The abstract transitions in Figure 5 can be used to define abstract transition functions. This is possible since the the transitions are deterministic. For a given program, a system of abstract data flow equations can be set up based on the control graph of the program (see Figure 3). Each abstract

$$\tilde{A}[\![n]\!]\tilde{\sigma} = \begin{cases} \mathit{NID} & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![var]\!]\tilde{\sigma} = \begin{cases} \tilde{\sigma}(var) & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![var[a]]\!]\tilde{\sigma} = \begin{cases} \tilde{\sigma}(var[\,]) & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ & \text{and } \tilde{A}[\![a]\!]\tilde{\sigma} = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\tilde{A}[\![a_1 + a_2]\!]\tilde{\sigma} = \tilde{A}[\![a_1 - a_2]\!]\tilde{\sigma} = \tilde{A}[\![a_1]\!]\tilde{\sigma} \sqcup \tilde{A}[\![a_2]\!]\tilde{\sigma}$$

$$\tilde{B}[\![\texttt{true}]\!]\tilde{\sigma} = \tilde{B}[\![\texttt{false}]\!]\tilde{\sigma} = \begin{cases} \mathit{NID} & \text{if } \tilde{\sigma}(\texttt{flow}) = \mathit{NID} \\ \mathit{ID} & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\tilde{B}[\![a_1 = a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 < a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 \leq a_2]\!]\tilde{\sigma} = \\ &\tilde{B}[\![a_1 > a_2]\!]\tilde{\sigma} = \tilde{B}[\![a_1 \geq a_2]\!]\tilde{\sigma} = \\ &\quad \tilde{A}[\![a_1]\!]\tilde{\sigma} \sqcup \tilde{A}[\![a_2]\!]\tilde{\sigma} \end{aligned}$$

$$\tilde{B}[\![\neg b]\!]\tilde{\sigma} = \tilde{B}[\![b]\!]\tilde{\sigma}$$

$$\tilde{B}[\![b_1 \wedge b_2]\!]\tilde{\sigma} = \tilde{B}[\![b_1]\!]\tilde{\sigma} \sqcup \tilde{B}[\![b_2]\!]\tilde{\sigma}$$

Figure 4. Abstract semantics of expressions

Assign to variable: $\langle var := a, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}[var \mapsto \tilde{A}[\![a]\!]\tilde{\sigma}]$

Assign to array: $\langle var[a_1] := a_2, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}[var[] \mapsto (\tilde{A}[\![a_2]\!]\tilde{\sigma} \sqcup \tilde{\sigma}(var[]))]$

If(*NID*): $$\frac{\langle S_1, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}'' \qquad \langle S_2, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}''' \qquad \tilde{\sigma}' = \tilde{\sigma}'' \sqcup \tilde{\sigma}'''}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}'} \quad \text{if } \tilde{B}[\![b]\!]\tilde{\sigma} = \mathit{NID}$$

If(*ID*): $$\frac{\langle S_1, \tilde{\sigma}[\texttt{flow} \mapsto \mathit{ID}] \rangle \Rrightarrow \tilde{\sigma}'' \qquad \langle S_2, \tilde{\sigma}[\texttt{flow} \mapsto \mathit{ID}] \rangle \Rrightarrow \tilde{\sigma}''' \qquad \tilde{\sigma}' = \tilde{\sigma}'' \sqcup \tilde{\sigma}'''}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}'[\texttt{flow} \mapsto \tilde{\sigma}(\texttt{flow})]} \quad \text{if } \tilde{B}[\![b]\!]\tilde{\sigma} = \mathit{ID}$$

While: $\langle \texttt{while}\ b\ \texttt{do}\ S, \tilde{\sigma} \rangle \Rrightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, \tilde{\sigma} \rangle$

Seq(1): $$\frac{\langle S_1, \tilde{\sigma} \rangle \Rrightarrow \langle S_1', \tilde{\sigma}' \rangle}{\langle S_1; S_2, \tilde{\sigma} \rangle \Rrightarrow \langle S_1'; S_2, \tilde{\sigma}' \rangle}$$

Seq(2): $$\frac{\langle S_1, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}'}{\langle S_1; S_2, \tilde{\sigma} \rangle \Rrightarrow \langle S_2, \tilde{\sigma}' \rangle}$$

Skip: $\langle \texttt{skip}, \tilde{\sigma} \rangle \Rrightarrow \tilde{\sigma}$

Figure 5. Abstract transitions

state corresponds to an edge in the the control graph, and each equation is based on an abstract transition function of a node (statement) in the graph. The initial abstract state is assigned to the input edge of the program.

Primitive transition functions. First we define the abstract transition functions for the primitive (i.e., not compound) statements, corresponding to the first line of Figure 3.

Assign to variable ($var := a$):

$$\tilde{\sigma}_2 = \tilde{\sigma}_1[var \mapsto \tilde{A}[\![a]\!]\tilde{\sigma}_1]$$

Assign to array ($var[a_1] := a_2$):

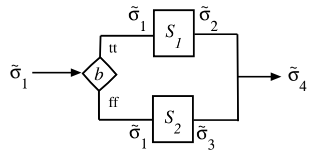
$$\tilde{\sigma}_2 = \tilde{\sigma}_1[var[] \mapsto (\tilde{A}[\![a_2]\!]\tilde{\sigma}_1 \sqcup \tilde{\sigma}_1(var[]))]$$

Skip:

$$\tilde{\sigma}_2 = \tilde{\sigma}_1$$

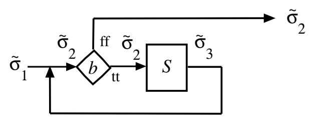
Compound transition functions. The compound functions are built up from the primitive functions. For the if and while statements we also need to define what happens in the merge point.

If ($\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$):



$$\tilde{\sigma}_4 = \begin{cases} (\tilde{\sigma}_2 \sqcup \tilde{\sigma}_3)[\texttt{flow} \mapsto \tilde{\sigma}_1(\texttt{flow})] & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_1 = \mathit{ID} \\ \quad \text{where} & \\ \quad \tilde{\sigma}_2 = \tilde{[\![}S_1\tilde{]\!]}\tilde{\sigma}_1[\texttt{flow} \mapsto \mathit{ID}] \text{ and} & \\ \quad \tilde{\sigma}_3 = \tilde{[\![}S_2\tilde{]\!]}\tilde{\sigma}_1[\texttt{flow} \mapsto \mathit{ID}] & \\ \tilde{\sigma}_2 \sqcup \tilde{\sigma}_3 & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_1 = \mathit{NID} \\ \quad \text{where} & \\ \quad \tilde{\sigma}_2 = \tilde{[\![}S_1\tilde{]\!]}\tilde{\sigma}_1 \text{ and } \tilde{\sigma}_3 = \tilde{[\![}S_2\tilde{]\!]}\tilde{\sigma}_1 & \end{cases}$$

While (`while` b `do` S):



$$\tilde{\sigma}_2 = \begin{cases} (\tilde{\sigma}_1 \sqcup \tilde{\sigma}_3)[\texttt{flow} \mapsto \tilde{\sigma}_2(\texttt{flow})] & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_2 = ID \\ \quad \text{where} & \\ \quad \tilde{\sigma}_3 = \tilde{[\![S]\!]}\tilde{\sigma}_2[\texttt{flow} \mapsto ID] & \\ \tilde{\sigma}_1 \sqcup \tilde{\sigma}_3 & \text{if } \tilde{B}[\![b]\!]\tilde{\sigma}_2 = NID \\ \quad \text{where} & \\ \quad \tilde{\sigma}_3 = \tilde{[\![S]\!]}\tilde{\sigma}_2 & \end{cases}$$

Seq (S_1;S_2):

$$\begin{aligned} \text{Seq}(1) : \tilde{\sigma}_2 &= \tilde{[\![S_1'; S_2]\!]}(\tilde{[\![S_1]\!]}\tilde{\sigma}_1) \\ \text{Seq}(2) : \tilde{\sigma}_2 &= \tilde{[\![S_2]\!]}(\tilde{[\![S_1]\!]}\tilde{\sigma}_1) \end{aligned}$$

where $\tilde{[\![S]\!]}$ denotes the abstract transition function of S.

Solution. We can solve the system of data flow equations by using Jacobi iteration, i.e., by iteration until a fixpoint is reached. Given a set of data flow equations f_i so that $\tilde{\sigma}_i = f_i(\tilde{\sigma}_1, \ldots, \tilde{\sigma}_n)$ for $i = 2, 3 \ldots n$ and defining F as

$$\begin{aligned} F(\tilde{\sigma}_1, \ldots, \tilde{\sigma}_n) &= (y_1, \ldots, y_n) \text{ where } y_1 = \tilde{\sigma}_0 \text{ and} \\ y_i &= f_i(\tilde{\sigma}_1, \ldots, \tilde{\sigma}_n) \text{ for } i = 2, 3 \ldots n \end{aligned}$$

we can obtain the least fixed point of F as the least upper bound of the ascending chain $\perp, F(\perp), F(F(\perp)), \ldots F^n(\perp)$. The chain is finite since the domain used is finite. Therefore, we are guaranteed to find a solution with a finite number of steps.

6 Example.

A simple example will show the use of the rules, especially the handling of the `flow` variable. Let's analyze the following program:

```
S = if x = 1 then z := 1 else z := 2;
if y = 1 then x := 1 else x := 2
```

with the initial state

$$\tilde{\sigma}_0 = [\texttt{x} \mapsto ID, \texttt{y} \mapsto NID, \texttt{z} \mapsto NID, \texttt{flow} \mapsto NID].$$

For this example, using the rules in Figure 5, we get the flow equations as shown in Figure 7. We have simplified some of the expressions by using rules from Figure 4. The abstract states in the equations refer to the corresponding states in Figure 6, which shows the control flow graph of the program.

Solution of the example. The Jacobi iteration of the system of flow equations in Figure 7 will reach a fixpoint after 9 iterations. The abstract states for two interesting program points are:

$$\tilde{\sigma}_6 = [\texttt{x} \mapsto ID, \texttt{y} \mapsto NID, \texttt{z} \mapsto ID, \texttt{flow} \mapsto NID]$$

The result of the analysis of the first if-statement is that `z` is set to *ID*. The reason for this is that the first condition `if x = 1` is input dependent (since `x` is). Therefore, `z` is regarded as input dependent (even if `z` is assigned constants in both edges!).

$$\tilde{\sigma}_{11} = [\texttt{x} \mapsto NID, \texttt{y} \mapsto NID, \texttt{z} \mapsto ID, \texttt{flow} \mapsto NID]$$

The result of the analysis of the entire program is that `z` is set to *ID* (as described above) and that `x` is set to *NID* due to the second condition `if y = 1` is not input dependent (since `y` is not). For the code generation, this means that the first if-statement must generate predicated code, while the second can generate ordinary code.

7 Conclusions and Future Work

In this paper we described an analysis technique that identifies input-data dependent control conditions in loops or branching statements of real-time code. This analysis is needed for two purposes.

First, tools using the analysis support the programmer in writing code that avoids input-data dependent control flow as far as possible.

Second, the single-path conversion technique relies on this analysis to remove input-data dependent control dependencies from the code. Both, avoiding and eliminating input-data dependent control flow are important to produce good real-time code, i.e., code with a small execution-time jitter for which safe and tight upper bounds on the execution-time can be computed.

The analysis of input dependencies is based on abstract interpretation. The **While** language was used to provide a formal description of the abstract-interpretation framework and demonstrate how it finds input dependencies. A number of examples was used to illustrate the approach.

As a next step we plan to extend the simple semantics used in this paper to a real imperative programming language like C.

Further, we will develop a tool that implements the analysis and analyzes real code. The tool will be realized as part of a compiler.

References

[AKPW83] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence

$S = S_1; S_2$

$S_1 =$ `if` b_1 `then` S_3 `else` S_4

$b_1 =$ `x = 1`

$S_3 =$ `z := 1`

$S_4 =$ `z := 2`

$S_2 =$ `if` b_2 `then` S_5 `else` S_6

$b_2 =$ `y = 1`

$S_5 =$ `x := 1`

$S_6 =$ `x := 2`

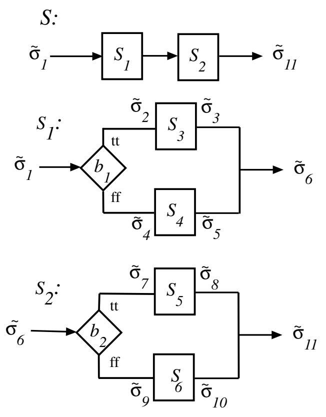


Figure 6. Control flow graphs for the example

$$\begin{aligned}
\tilde{\sigma}_1 &= \tilde{\sigma}_0 \\
\tilde{\sigma}_2 &= \begin{cases} \tilde{\sigma}_1 \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \\ \tilde{\sigma}_1[\mathtt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \end{cases} \\
\tilde{\sigma}_3 &= \tilde{\sigma}_2[\mathtt{z} \mapsto \tilde{A}[\![1]\!]\tilde{\sigma}_2] \\
\tilde{\sigma}_4 &= \begin{cases} \tilde{\sigma}_1 \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \\ \tilde{\sigma}_1[\mathtt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \end{cases} \\
\tilde{\sigma}_5 &= \tilde{\sigma}_4[\mathtt{z} \mapsto \tilde{A}[\![2]\!]\tilde{\sigma}_4] \\
\tilde{\sigma}_6 &= \begin{cases} (\tilde{\sigma}_3 \sqcup \tilde{\sigma}_5)[\mathtt{flow} \mapsto \tilde{\sigma}_1(\mathtt{flow})] \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{ID} \\ \tilde{\sigma}_3 \sqcup \tilde{\sigma}_5 \text{ if } \tilde{B}[\![\mathtt{x} = \mathtt{1}]\!]\tilde{\sigma}_1 = \mathit{NID} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\tilde{\sigma}_7 &= \begin{cases} \tilde{\sigma}_6 \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \\ \tilde{\sigma}_6[\mathtt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \end{cases} \\
\tilde{\sigma}_8 &= \tilde{\sigma}_7[\mathtt{x} \mapsto \tilde{A}[\![1]\!]\tilde{\sigma}_7] \\
\tilde{\sigma}_9 &= \begin{cases} \tilde{\sigma}_6 \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \\ \tilde{\sigma}_6[\mathtt{flow} \mapsto \mathit{ID}] \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \end{cases} \\
\tilde{\sigma}_{10} &= \tilde{\sigma}_9[\mathtt{x} \mapsto \tilde{A}[\![2]\!]\tilde{\sigma}_9] \\
\tilde{\sigma}_{11} &= \begin{cases} (\tilde{\sigma}_8 \sqcup \tilde{\sigma}_{10})[\mathtt{flow} \mapsto \tilde{\sigma}_6(\mathtt{flow})] \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{ID} \\ \tilde{\sigma}_8 \sqcup \tilde{\sigma}_{10} \text{ if } \tilde{B}[\![\mathtt{y} = \mathtt{1}]\!]\tilde{\sigma}_6 = \mathit{NID} \end{cases}
\end{aligned}$$

Figure 7. Data flow equations for the example program

to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Gus00] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications*. John Wiley & Sons, 1992.

[PB02] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[Pus02] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002.

[Pus03] Peter Puschner. Algorithms for dependable hard real-time systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.