Intelligent Editor for Writing Worst-Case-Execution-Time-Oriented Programs *

Janosch Fauster, Raimund Kirner, and Peter Puschner

Institut für Technische Informatik,
Technische Universität Wien,
Treitlstraße 3/182/1,
A-1040 Wien, Austria

Abstract. To guarantee timeliness in hard real-time systems the knowledge of the worst-case execution time (WCET) for its time-critical tasks is mandatory. Accurate and correct WCET analysis for modern processors is a quite complex problem. Path analysis is required to identify a minimal set of possible execution paths. Further, the modeling of a processor's internal states for features like caches or pipelines requires to consider possible interferences of these features.
This paper presents a new software engineering paradigm tailored to the development of real-time software. This paradigm results into more predictable programs and is therefore well-suited for the development of real-time systems. New software development tools are necessary to support developers in writing efficient code for this new paradigm. In this paper an editor is described that highlights all code that is not conform with this programming paradigm.

1 Introduction

The knowledge of the worst-case execution time (WCET) of tasks is crucial for the design of real-time systems. Only if safe upper bounds for the WCET of all time-critical tasks have been established, it becomes possible to verify the timeliness of the whole real-time system. Over the last one-and-a-half decades research in WCET analysis and real-time computing has solved many sub-problems of WCET analysis. Despite this progress in WCET analysis, there still exist three fundamental problems in the current state of the art in WCET analysis [8]:

First, WCET analysis needs exact knowledge about the possible execution paths through the analyzed code. Deriving this information automatically is, however, not possible in the general case. This is due to the fact that the control flow of a program typically depends on the input data of the program and a WCET bound thus cannot be predicted purely from code analysis. Further,

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

the fully automatic program analysis to derive descriptions about possible control flows automatically is in conflict to the halting problem. Therefore, current WCET analysis tools rely on the provision of the lacking path information [2, 3].

The second major problem is obtaining correct and accurate models about the timing behavior of modern processors. These processors typically use features like caches or pipelines to improve their peak-performance. The effects of these hardware features interfere with each other and are therefore hard to predict. Even worse, the behavior of a processor generally is scarely documented [1]. These facts taken together make it difficult if not impossible, to build for WCET analysis tools a correct and accurate hardware model of the target processor.

The third major problem is the complexity of the WCET analysis. Beside the problems in identifying the possible execution paths and obtaining detailed hardware-timing data, the complexity of WCET analysis itself is a problem. The number of paths that have to be analyzed to calculate a precise WCET bound is growing exponentially with the number of consecutive branches. Full path enumeration therefore becomes infeasible, except for programs having a very simple control flow [4]. To overcome this problems, approximating analysis techniques are used. These approximations causes overestimation and consequently lead to a system design with decreased utilization of hardware resources.

A possible solution to the above problems is the use of new software engineering paradigms tailored to the development of real-time software. A recently developed paradigm for this new area of software engineering is WCET-oriented programming [8, 7]. It represents an unconventional way on how to write programs. The fundamental motivation of WCET-oriented programming is to reduce the number of program statements with input-data dependent control flow. New software development tools are necessary to support software developers in writing efficient programs for this new software engineering paradigm.

In this paper we describe an editor that is able to highlight code that is not conform with this programming paradigm. The described analysis method to highlight the code is integrated as plug-in into a popular editor.

This article is structured as follows: Section 2 discusses WCET-oriented programming and introduces the *single-path approach* to increase predictability in real-time programs. Section 3 describes an analysis method to detect program statements causing an input-data dependent control flow. The integration of this analysis into an editor is explained in section 4. Examples are shown in section 5 to demonstrate the features of this program analysis methods. Section 6 gives a conclusion to the article.

2 Writing WCET-Oriented Programs

WCET-oriented programming is a software engineering paradigm especially tailored to the development of real-time software. This section discusses its novel aspects compared to traditional performance-oriented programming. Further,

the *single path approach* – a paradigm to increase predictability of real-time software – is described.

2.1 Traditional Performance-Oriented Coding

Non real-time programmers typically aim at a good average performance to allow for a high throughput. Therefore, the primary performance goal of non real-time programmers is the speed optimization for the most probable (i.e., frequent) scenarios. In order to be able to favor the frequent cases the code tests the properties of input-data sets and chooses the actions to be performed during an execution based on input data. Using input-data dependent control decisions is an effective way to achieve short execution times for the favored input-data sets. This approach is therefore suitable for optimizing the average execution time. In contrast to this, a programming style that is based on input-data dependent control decisions adversely affects the quality of the achievable WCET. This is due to the following reasons:

- *Tests to identify the current input data*: Even if an input-data set is not among the "favored" inputs it has to be tested at the points where the control flow between favored and non favored inputs splits. While the fast code makes up for the cost of the control decisions in the case of favored inputs, the execution time of the input-data tests add up to the execution time without compensation for all other data.
- *Branching costs*: Similarly to the previous argument, not just the costs for testing the properties of input data but also the costs for branching to the respective code sections increase the total execution time of non-favored cases.
- *Information-theoretical imbalance*: Every functionality on a defined input-data space and available data memory has a specific complexity. The overall problem complexity determines the number and types of operations needed to solve the problem for the given input-data space. Performance oriented, non real-time programming spreads this overall complexity unevenly over the input-data scenarios: to facilitate a high throughput, the frequent input-data scenarios are treated at computational cost that are below the complexity that would result if the total complexity would be evenly distributed to all scenarios. As the complexity inherent to a problem is constant, a cost reduction for some part of the input-data space necessarily causes higher costs for the rest of the inputs. Again, this impairs the achievable WCET (example: average versus worst-case transmission time of a string coded in Huffman code respectively a constant-length code).

Data-dependent control decisions are the results of traditional performance-optimization patterns. In the following we show that we have to apply a completely different and not so common optimization strategy, if we aim at optimizing the worst-case completion time.

2.2 Programming for the Worst-Case

As shown in the previous section, traditional (non real-time) programming tends to produce code that has a high WCET. We observe that it is the different treatment of scenarios, i.e., favoring certain input-data sets over others, that causes an increased WCET. The reasons for this were detailed above. In order to write code that has a good WCET the shortcomings of the traditional programming style have to be avoided. A novel programming strategy is needed. 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 operations that are only executed for a subset of the input-data space to a minimum. Note that in some applications it is impossible to treat all inputs identically. This can be due to the inherent semantics of the given problem or the limitations of the programming language used. WCET-oriented programming needs a way of thinking that is quite different from the solution strategies we normally use. As a consequence, it produces unconventional algorithms that may not look straightforward at the first sight. The resulting pieces of code, however, are characterized by competitive WCETs due to the small number of tests (and branches) on input data and the minimal information-theoretical imbalance. A small number of input-data dependent alternatives does not only keep the WCET down. It also keeps the total number of different execution paths through a piece of code low. Identifying and characterizing a smaller number of paths for WCET analysis is easier and therefore much less error-prone than dealing with a huge number of alternatives. In this way, WCET-oriented programming does not only produce code with better WCET performance but also yields more dependable WCET-analysis results and thus more dependable real-time code than traditional programming.

2.3 The Single-Path Approach for Predictable Code

As mentioned before, the problem of WCET analysis is in general a complex task because programs behave differently for different input data, i.e., different input data cause the code to execute on different execution paths with differing execution times. In the following we propose an approach that avoids this complexity by ensuring that the code to be WCET-analyzed has only a single execution path. This approach uses code transformations to transform input-data dependent branches and their alternatives into sequential code (input-data independent branches are not transformed). To be precise, the code resulting from the transformation avoids data dependencies in execution times by keeping input-data dependent branching local to single operations with data-independent execution times.

Constant-Time Conditional Expression The key feature of our predictable-programming approach is the so-called constant-time conditional expression operation (CTCE). A CTCE consists of a boolean expression and two expressions.

It evaluates the two expressions and returns one out of the two results. Which of the two results is actually selected depends on the truth value of the condition. Throughout this paper we use the following notation for CTCEs:

$$cond \ \# \ expr_1 \ : \ expr_2$$

cond represents the condition of the constant-time conditional, $expr_1$ and $expr_2$ stand for the two expressions that are evaluated. If *cond* yields true then the value of the CTCE is the result of $expr_1$. If *cond* evaluates to false the CTCE returns the value of $expr_2$. What has been described above may remind the reader of the conditional assignment operator "?:" of the C programming language. Indeed, as we assume that $expr_1$ and $expr_2$ do not have any side effects, the final result of both types of statements is the same. That is also why the syntax of the CTCE has been chosen similar to that of the C conditional expression (?:). Note, however, that there are significant differences in the control flow of the two constructs. The C conditional expression works like an *if-then-else* construct. It first evaluates the condition and then executes one of the two branches. In contrast to the conditional expression in C, the CTCE evaluates both expressions. Having computed the expressions it evaluates the condition and returns one of the two expression results as the value of the whole constant-time conditional expression. In [8] the different semantics of the two conditional operators are described in more detail.

Obviously, evaluating a conditional expression with the new operator takes longer than using an operator with the semantics of the C construct (the old operator evaluates only one expression while the new one has to evaluate both). The big advantage of the CTCE shows, however, when it comes to execution time prediction. In the C conditional the two alternatives of the branch perform different operations, and this usually implies that the alternatives have different execution times. In the general case it is therefore impossible to predict the exact execution time of the conditional. The constant-time implementation has a single, constant execution time. Its execution time is therefore predictable. This is achieved as follows: First, both alternative expressions execute in sequence. Executing both expressions unconditionally avoids the problem of finding out how the conditional behaves in a worst-case execution. Second, the result of the overall conditional is selected and returned in a simple operation with constant execution time.

The CTCE can be realized with a *conditional move instruction*, which is implemented on a number of modern processors [8].

2.4 Converting WCET-Analyzable Code into Single-Path Code

The CTCE is a construct to make the *single-path approach* explicit. In the following we illustrate how every well structured and WCET-analyzable piece of program code can be translated into code with a single execution path. By the term WCET-analyzable code we understand code for which the maximum number of iterations of every loop is known; a WCET bound is thus computable.

The translation replaces all input-data dependent branches by sequential code that uses CTCEs [6].

We only consider structured data dependent branchings of high-level languages like conditional statements (e.g., if statement) and loops; gotos or exit statements are leaved out. In order to translate a piece of code into temporally predictable code we transform these two statement types into non-branching code.

- Conditional branching statements conditionally change the values of a number of variables. The transformation of such conditional branches is straightforward. The translation process generates sequential code with constant-time conditional assignments for each of the conditionally changed variables. When translating assignments in nested conditional branches, the conditions of all nested branches have to combined in the conditions of the generated conditional assignments.
- Loops with input-data dependent termination conditions are translated in two steps. First, the loop is changed into a simple counting loop with a constant iteration count. The iteration count of the new loop is set to the maximum iteration count of the original loop. The old termination condition is used to build a new branching statement inside the new loop. This new conditional statement is placed around the body of the original loop and simulates the data dependent termination of the original loop in the newly generated counting loop. The second step of the loop translation transforms the new conditional statement, that has been generated from the old loop condition, into a constant-time conditional assignment. This way the entire loop executes in constant time.

Note that applying the described transformation to existing real-time code may yield temporal predictability at a very high cost in terms of execution time. Thus, we consider the illustration of the transformation as a demonstration of the general applicability of our approach, rather than proposing to use the transformation for generating temporally predictable code from arbitrary real-time programs.

In order to come up with code that is both temporally predictable and well performing the programmer needs to use adequate algorithms, i.e., algorithms with no or minimal input-data dependent branching. This new paradigm tailored to the development of real-time software is called WCET-oriented programming. Special software engineering tools can help the software developer in writing WCET-oriented software. In the following sections a special feature for an editor is described that provides the developer with additional information about the predictability of the code. Statements that cause an input-data dependent change of the control flow (i.e., statements that violates the single-path approach) are highlighted. This information allows the developer to check the predictability of the current code. Based on this information, a refinement of the code may be possible to avoid any performance decrease caused by the automatic translation of the program into single-path code.

3 Control Flow Analysis

We developed a plug-in for an editor to support the development of WCET-oriented code. This plug-in can analyze the source code and mark every control-flow that violates the *single-path paradigm*.

In order to analyze the source code, our first task is to detect all *basic blocks* in the code and to construct a *control flow graph* representing all control flows between the basic blocks. The second step is to extract data information from input code and to analyze the *data flow* in the function. For that reason we need a data structure that supports a correct and efficient analysis of the code. The data flow information we need, can be represented as a *semi-lattice*, where the elements of the lattice constitute abstract properties of the program. In our case each variable is mapped to a lattice shown in figure 1 containing the following three elements:

- the *bottom* element ($\perp$) that marks an "unreachable value"
- *maybe input dependent* (MBID) that marks a variable as "possibly input dependent" and finally
- *not input dependent* (NID) that marks an element as surely "independent from the input"

Fig. 1. Hasse Diagram of Lattice

This semi-lattice induces a partial order on its elements. The diagram represents the information content of the values, i.e. upper values have a less precise information content than lower values.

The initial state of the semi-lattice is NID for all elements in all basic blocks. But there is an exception. The *start* node contains all the informations that we have at the beginning of the algorithm and so it gets initialized as follows:

- all global variables are marked as "maybe input dependent", because we consider only intra-procedural control flows and so we don't know what happens outside a function.
- all parameters of the function are considered as input dependent too. This is evident, because these parameters are the input for the function.
- all locally defined variables are initialized as "not input dependent".

After this initialization we have a complete control flow graph and we can solve the problem with a modified fixpoint iteration scheme. For that purpose we have developed an analysis algorithm which is subdivided into three levels:

1. basic block level analysis
2. statement level analysis
3. expression level analysis

Each of this steps will now be covered in detail.

3.1 Basic Block Analysis

First of all we take an arbitrary basic block out of the control flow graph. Each basic block has an input vector and an output vector, in which the state (MBID, NID or $\perp$) at the entry point and at the exit point of the basic block for all variables is saved. Then we have to compute the current input vector. If a basic block can be reached by only one other basic block, we simply set the input vector to the output vector of that basic block. Otherwise, if the basic block can be reached by more than one control path, we have to apply a union operation on all output vectors of the preceding basic blocks in order to get the input vector of the current node. This union operation is defined in the following way:

$$\bigcup\nolimits_{bb}(p_1 \dots p_n) = \begin{cases} \perp & \text{if } \forall j \in \{1 \dots n\} : p_j = \perp \\ \text{MBID} & \text{if } \exists j \in \{1 \dots n\} : p_j = \text{MBID} \\ \text{NID} & \text{otherwise} \end{cases} \quad (1)$$

which means that if the state of a variable is NID in all vectors, it remains NID in the input vector, but if the state in only one of the considered output vectors is MBID, the new value is MBID.

After we have computed the input vector, all statements and all expressions in these statements are analyzed. Details regarding those analyses are given in the next sections. The whole procedure for calculating the output vectors of all basic blocks in the control flow graph is repeated until the output values are stable for all nodes. It can be shown that this algorithm always terminates.

3.2 Statement Analysis

As mentioned before, each statement in a basic block must be analyzed separately. It is important to notice, that a vector is associated to each statement, in which the state of all variables in that particular code location is stored. So the first task is to update the current vector. If the statement to be analyzed is the first in the basic block it takes over the input vector of the basic block, otherwise it uses the vector of the statement preceding it.
The statement analysis module mainly extracts all expressions from the statement and passes them to a dedicated expression analyzer. This is true for simple arithmetic expressions, but also for the conditional expressions in while-,

```
main()
{
        int b;
        if (a)
                b=2;
        if (b)
                doSomething;
}
```

Fig. 2. Code Example for Indirect Flow Dependency

do/while-, for-, if/else- and switch/case-statements. The second job for this module is to detect indirect control flow dependencies.

The example in figure 2 shows such an indirect dependency. The first if-statement in the illustration depends on a global value. That means, that the assignment-expression `b=2` is also input dependent, although `b` is assigned only a constant. That means furthermore that also the second if-statement depends on global input data and should be marked accordingly.

3.3 Expression Analysis

This is the central part of the analysis, because here we actually set the state of the variables. Each expression has a set of input parameters $in = \{in_1 \dots in_n\}$ and a set of output parameters $out = \{out_1 \dots out_n\}$, where it is possible to deduce the state of all output parameters from the state of the input parameters using following function:

$$\forall out_i \in out : \\ out_i = \bigcup_{expr}\{in_1 \dots in_n\} = \begin{cases} \perp & \text{if } \forall j \in \{1 \dots n\} : in_j = \perp \\ \text{MBID} & \text{if } \exists j \in \{1 \dots n\} : in_j = \text{MBID} \\ \text{NID} & \text{otherwise} \end{cases} \quad (2)$$

We distinguish the following cases:

- *assignments*, like `a=b+c`, ($in = \{b, c\}$, $out = \{a\}$).
  All elements at the right side of the assignment are input elements, the variable at the left side is the output element. This means, that if all elements at the right side of the assignment are NID, then also the variable at the left side is NID. But if at least one element at the right side has the value MBID, then also the variable at the left side gets MBID.
- *functions*. in = {MBID}, out ={global variables, return-value, referenced values of pointers in arguments}.
  Functions need a more complicated handling. At this point it is important to remember that we make only intraprocedural and no interprocedural analysis. Thus, after a function call we must assume, that all globally declared

variables could have been changed during the function call and for that reason the value of all those variables is set to MBID. Further, the return value of the function is always input dependent. Finally, another case must be considered: if one of the arguments of the called function is a pointer, its referenced value is set to MBID, too.

- *constants* ($in = \emptyset$, $out = \{\text{constants}\}$).
  Constants are considered to be always NID.
- *structures/unions*. Each member of a structure is treated separately. Thus a single member can have the value MBID, while other ones are in the state NID. Special care must be taken for recursive structures, e.g. datastructures used for trees and lists. Here we restrict the accuracy of the analysis to assure the correctness of the algorithm. We assume that each locally defined structure is NID at the beginning. But if at least one member gets MBID, the whole structure turns to MBID and cannot be changed any longer to NID.
- *pointers*. Each pointer is represented by two state values. One for the pointer itself and a second one for the referenced variable. The pointer itself can be treated like a common variable, but the referenced variable needs special attention, because it could be referenced by more than one pointer. For that purpose we have two different strategies. The easier one assumes that all referenced variables have always MBID as their current state, while the more sophisticated strategy uses alias-informations to handle references by pointers. More informations about the alias-analysis can be found in section 3.4.
- *arrays*. Arrays are handled similarly as pointers, i.e. each array is represented with two states, one for the array itself and one for *all* its references. Thus, if one array entry changes its state from NID to MBID, the whole array is considered to be MBID and remains in this states forever. This is necessary because the analysis doesn't keep track of individual array cells and so if one value changes its state to MBID, later we don't know which cell was affected by this change and thus we must assume, that all cells could be in the state MBID. We have chosen this way to handle arrays, because it's quite simple, whereas developing a method that considers each element of an array individually would be complex and leads to higher computation and memory consumption. Arrays can be subject to aliasing too and therefore we use our aliasing analysis with arrays as well.

3.4 Alias Analysis

One or more pointers can reference the same memory location. If one pointer changes the value of that location, all other pointers that reference the same location will see this new value, too. Thus we must ensure that our analysis takes into consideration this behavior. As mentioned before, a simple strategy would be that all referenced variables are always set to MBID. By assuming the "worst case", we can be sure that our analysis will always be correct. In most cases this treatment of pointers (and arrays) is sufficient, but there are other cases were a more accurate view is needed. For more accurate results we

have developed an alias analysis, which recognizes pointers that could reference the same memory location. To keep things simple, we have put a restriction on our implementation: all global variables (including globally declared pointers) are assumed to refer to the same memory location. In most cases this will not be true, but our analysis is "safe", i.e. if an element is set to NID it is surely not input dependent, whereas if it is set to MBID it *may* depend on the input values. Following this restriction, all these global variables have to be marked as "aliased".

```
int a;
main()
{
        int *x, *y, *z;

        y=z;
        x=y;

        *z=a;
}
```

Fig. 3. Example for Aliasing Analysis

The example in figure 3 shows a typical case, where an alias analysis should be used. The first statement (`y=z`) means that both `y` and `z` reference the same memory location and thus have to be aliased. The result is shown in figure 4a, where each variable has its own list containing those variables to which it could be aliased (for a more detailed description of the used data structures refer to section 4).

a) aliased variables 1 b) aliased variables 2

Fig. 4. Aliased Variables for the Example Code

The second statements adds further aliasing information. After the execution of the statement (`x=y`), `x`, `y` and `z` are aliased. The result is shown in figure 4b.

Now, the last statement in the example will need the collected aliasing informations to produce a correct result. Here the referenced variable of the pointer `z` gets a new, input dependent, value and so it is set to MBID. Now the analyzer sees that aliasing information is connected with this variable and therefore up-

dates also the states of the other two pointers. At the end of this piece of code the output vector which stores the input dependency information is shown in figure 5.

	a	x	*x	y	*y	z	*z
MBID	1	0	1	0	1	0	1

Fig. 5. Flow Dependency Output Vector

4 Editor with Integrated CFA

We have implemented a plug-in for an editor that supports the data flow analysis described in the previous chapter. By using this enhanced editor the programmer has two possibilities:

1. writing programs in such a manner that nothing will be highlighted. With this proceeding one can be sure, that the program is conform to the *single path approach*.
2. often the first procedure is not possible, because a program already exists in large parts or the restrictions implied to are too big. In this case the advanced features of the editor are use to highlight all code that could violate the *single path approach* and one can verify, whether they are really critical or still tolerable.

We have implemented a plug-in for *vim*[1], that is capable of analyzing ANSI C89 code, but for more programming conveniences it supports also many ANSI C99 [5] and GNU C-expansions [9], for example additional datatypes and the //-comment. The CTCE, which is described in section 2.3, is also supported by the analysis. It has the following syntax:

`<expr> = <cond> ♯ <expr1> : <expr2>.`

This expression is translated into two conditional expressions, where `<expr1>` or `<expr2>` is assigned to `<expr>`. According to the *single path approach* `<expr1>` and `<expr2>` have to be free of side effects. In languages with pointers its very difficult to locate such side effects and so we put the following constraints on these expressions:

`<expr1>, <expr2>` $\in$ {`CONST, VAR`},

where `CONST` stands for a numerical constant (independent from the type) and `VAR` is an identifier representing a variable name.

The plug-in for *vim* is written in the vi scripting language. It executes the following steps:

[1] `http://www.vim.org`

– First of all it starts an external program, which analyzes the source code. This external program writes the results of the analysis, i.e. those line numbers in which there is code that violates the *single path approach*, in a file.
– Now we use a small program called "sequencer", which extracts from this file and returns the next line to be highlighted.
– We use the built-in features of vim to highlight this line.
– The sequencer-program is called in a loop until all flow dependent lines are highlighted.

The program that analyzes the code reconstructs a control flow graph after parsing the code. Each node in the control flow graph contains its assigned statements and pointers to its successor and predecessor nodes. Besides each node has two bitvectors storing the states of all variables at the entry and exit point of the basic block (`input` and `output` vectors). The dataflow analysis was implemented using the three-level algorithm presented in section 3. At statement level the detection of indirect control flow dependencies in nested loops was implemented using a global stack storing flags. If the flag at the top of the stack is set, the analyzer knows that it is currently inside a conditional expression dependent on the input value. At expression level we have some more datastructures: a *current vector* that stores the state of all variables after the execution of that expression and a field of lists storing the collected alias-informations by using the algorithm presented in section 3.4. The datastructure for storing aliasing information was initially implemented as a simple bit-matrix, where an entry x/y is true if the variables x and y are aliased. Tests have shown that this matrix could become large in functions with many variables, but is typically only sparse. Therefore, we decided to implement that matrix in form of a vector with references to simple lists. Each variable has its own list, where each list entry represents a variable to which this variable is aliased.

An expression is analyzed by going recursively to the innermost subexpression. This one is inspected and the results are passed back to the surrounding expression. By using the results of all subexpressions, the current expression can be examined. This mechanism is repeated until the whole expression is analyzed. After all variable states are stable, i.e. the output vector of all nodes does not change for a whole iteration, the algorithm terminates by collecting the results and highlighting them in the editor.

5 Examples

The examples given in this section illustrate the software engineering paradigm of WCET-oriented programming and demonstrate the program analysis to test whether a code is conform to this paradigm.

5.1 Example for WCET Oriented Programming

WCET-oriented programming is intended to increase the predictability of real-time programs. Figure 6a) shows a traditional performance-oriented implementation of find_first. Its behavior is that the loop is left as the first occurrence of

a key value is found within the array. In contrast, the WCET-oriented implementation shown in figure 6b) has an almost input-data independent runtime behavior. Its only control-flow variance comes from the (?:) operator. Therefore, also conventional code generated for this implementation has a reduced variance in the execution time. As shown in [7], for processors with hardware support for conditional move instructions the execution time becomes data-independent and its WCET is even better than the WCET for the traditional performance-oriented implementation.

```
int find_first
  (int key, int a[])
{
  int i;
  int pos = 100;

  for (i=0; i<=SIZE-1;i++)
  {
    if (a[i] == key)
    {
      pos = i;
      break;
    }
  }
}
```

a) Standard version

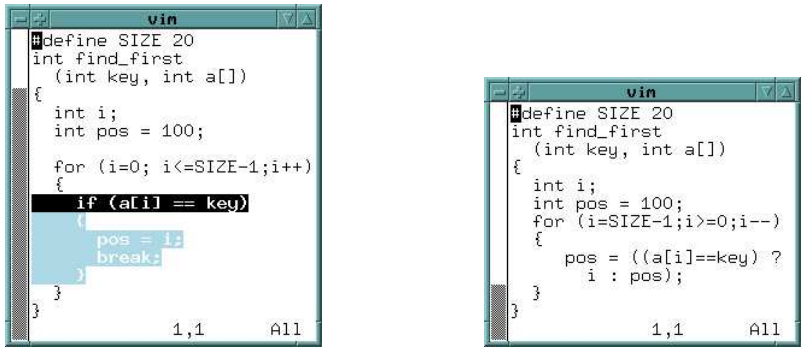
```
int find_first_wcet
  (int key, int a[])
{
  int i;
  int pos = 100;

  for (i=SIZE-1;i>=0;i--)
  {
    pos = ((a[i]==key) ?
      i : pos);
  }
}
```

b) WCET oriented version

Fig. 6. Example Code: find_first

The result of the conformance analysis for the WCET-oriented paradigm is shown in figure 7. The result for the traditional performance oriented implementation given in figure 7a) shows that there exists an input-data dependent control-flow path. The result for the WCET-oriented implementation given in figure 7b) shows that there does not exist any input-data dependent control-flow path. A technical detail is that in case the (?:) operator given in figure 7b would contain other assignments than simple numeric expressions, the statement would be classified as input-data dependent control flow.

The runtime behavior of the WCET-oriented implementation is therefore more predictable than the traditional performance-oriented implementation. Information like this can be used by the software developer to implement real-time code with increased predictability.



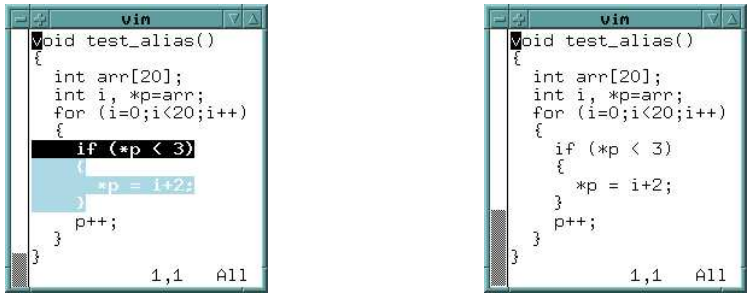
a) traditional performance-oriented implementation

b) WCET-oriented implementation

Fig. 7. Input Dependency Analysis for find_first

5.2 Example for Alias Analysis

The capability of the integrated alias analysis is shown by a simple example given in figure 8. The code in this example iterates over a local array having an input-data independent content.



a) without alias analysis

b) with alias analysis

Fig. 8. Input Dependency Analysis with Alias Information

As shown in figure 8a the conformance analysis for the WCET-oriented paradigm will classify the if statement as input-data dependent. The result for the conformance analysis with enabled alias analysis shown in figure 8b detects that this conditional control flow based on a pointer reference is input-data independent.

Due to the inherent complexity of an alias analysis for a programming language like C, our implementation is not able to precisely detect every input-data independent control-flow. But the alias analysis is safe in the sense that no input-data dependent control-flow will be classified as input-data independent.

6 Summary and Conclusion

To guarantee the timeliness of hard real-time systems it is necessary to perform WCET analysis for their time-critical tasks. Programs can be easily analyzed for their WCET, if they are translated into *single-path code*. To increase the efficiency of the translated code, a new programming paradigm – called WCET-oriented programming – has been developed. New software development tools are necessary to support developers in writing efficient code for this new paradigm.

In this paper we presented a method based on control and data flow analysis to analyze the conformance of a code with the *single-path approach*. The analysis has been implemented as a plug-in for the editor *vim*.

Experiments have been done to illustrate the application of the conformance analysis to sample algorithms. The results of the implementations for the traditional performance-oriented programming and WCET-oriented programming have been compared to demonstrate the improved predictability of WCET-oriented programming.

References

1. Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, Dec. 2001.
2. Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, May 2000.
3. Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
4. Thomas Lundqvist and Per Stenström. Timing analysis in dynamically scheduled mircoprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, Dec. 1999.
5. American National Standards Insitute/International Standards Organisation. *ISO/IEC 9899:1999 Programming Languages – C*. American National Standards Institute, New York, USA, 2 edition, Dec. 1999.
6. 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. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).
7. Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.
8. Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.
9. Richard Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. iUniverse.com, Inc., USA, 2000. gcc-2.96, ISBN 0-595-10035-X.