# Transforming Execution-Time Boundable Code into Temporally Predictable Code

Peter Puschner
*Institut für Technische Informatik, Technische Universität Wien, Austria*

**Abstract**: Traditional Worst-Case Execution-Time (WCET) analysis is very complex. It has to deal with path analysis, to identify and describe the possible execution paths through the code to be analyzed, and it has to model the worst-case timing of the possible paths on the target hardware. The latter is again non-trivial due to interference of modern hardware features like instruction pipelines, caches, and parallel instruction-execution units on the processor.

To simplify WCET analysis we have proposed a new programming paradigm, single-path programming. Every program following this paradigm has only a single possible execution path, which makes path analysis and thus WCET analysis almost trivial. In this work we show how any real-time program code that is WCET-analyzable can be transformed into single-path code. This demonstrates that the single-path paradigm provides a universal solution to simplify WCET analysis.

**Key words**: real-time systems, programming paradigms, worst-case execution time, computer architectures

## 1. INTRODUCTION

Knowing the worst-case execution time (WCET) of tasks is crucial for building real-time systems. Only if safe WCET bounds for all time-critical tasks of a real-time system have been established can the correct timely operation of the whole real-time computer system be verified. During the last decade many research groups have undertaken research on (static)

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35599-3_29

WCET analysis of real-time tasks and many sub-problems of WCET analysis have been solved, see (Puschner and Burns, 2000).

Despite the numerous efforts and results in WCET-analysis research there are still three significant obstacles to a safe and exact WCET analysis:

1. *Limits of automatic path analysis*: To compute a tight WCET bound, WCET analysis needs exact knowledge about the possible execution paths through the analyzed code. Deriving this information automatically is, in general, not possible: First, the control flow of a program typically depends on the input data of the program. Thus a WCET bound cannot be predicted purely from code analysis but needs additional information about possible input data or about the effects the possible input data have on the control flow. Second, the fully automatic program analysis would be in conflict to the halting problem. In order to allow for a WCET analysis despite these fundamental limits, current WCET tools rely on the user to provide the lacking path information (Colin and Puaut, 2000; Engblom and Ermedahl, 2000). Deriving this path information is an intellectually difficult, time-consuming, and error-prone task.
2. *Lack of hardware-timing data*: Modern processors use speed-up mechanisms like caches, instruction pipelines, parallel execution units, and branch-prediction to enhance execution performance. These mechanisms are complex in their implementation and have mutual interferences in their timing. Besides , the mechanisms and their timing are generally scarcely documented to protect the manufacturer's intellectual property (Petters and Färber, 1999). These facts taken together make it difficult if not impossible to build reliable tools for static WCET analysis of modern processors.
3. *Complexity of analysis*: It has been shown that the number of paths to be analyzed for an exact WCET analysis grows exponentially with the number of consecutive branches in the analyzed code when this code is to be executed on modern processors. Except for very simple programs this high complexity makes the full path enumeration needed for an exact WCET analysis intractable (Lundqvist and Stenström, 1999). WCET analysis therefore has to do with pessimistic approximations. These approximations, however, over-estimate WCET and make a certain waste of resources at runtime unavoidable.

In a previous paper (Puschner and Burns, 2002) we presented the single-path paradigm, a radical programming paradigm that avoids the problems of WCET analysis. Using this paradigm, programmers write programs whose behaviour is independent of input data. These programs only have a single execution path that is executed in each program execution. The execution time of programs written in the praradigm is therefore exactly predictable.

The fact that programs only have a single execution path makes WCET analysis trivial: First, path analysis is superfluous – observing the execution path of a code execution with any input data yields the singleton execution path. Second, there is no need for static WCET analysis. If programs only have a single path, as proposed, this singleton path is necessarily the worst-case path. Thus, obtaining the WCET by "exhaustive" measurements (either on the target or on a cycle-accurate hardware simulator) is possible. There is no need to build highly sophisticated tools for static analysis as this is the case for traditional code where the high number of input-data dependent test cases makes measurement-based WCET analysis infeasible.

Another property of the single-path approach is that the execution time of single-path code is free of jitter. Thus it is not necessary to introduce delay constructs into the code if a constant execution time of the code is required. Also, the timing analysis of multiple tasks with communication or synchronization constraints gets less complex if the execution times of the code between communication and synchronization points is fixed as opposed to the case when code execution times are variable.

An approach that only permits programs with a single execution path may seem to allow programmers to write only very simple programs. In this paper we demonstrate that the proposed approach is not at all that restrictive. In fact, we show how any piece of code that is WCET-analyzable can be translated into single-path code. The translation builds on if-conversion (Allen et al., 1983) to produce code that keeps input-data dependent alternatives local to single conditional operations with data-independent execution times. While if-conversion only translates branching code within innermost loops, the here-presented conversion also converts loops – all loops with input-data dependent termination conditions are translated into loops with constant iteration counts.

The paper is structured as follows: Section 2 explains the terms and assumptions used throughout the paper. Section 3 introduces the main concepts of the single-path approach. Section 4 explains the translation rules that are needed to translate WCET-analyzable programs into programs with a single execution path. Section 5 provides a program example and shows its conversion into single-path code. Section 6 gives a summary and conclusion.

## 2. TERMS AND ASSUMPTIONS

We view a program as a piece of code that defines the transformation of an initial store (assignment of values to all program variables) into a new, final store. The valid set of initial stores for an application is assumed to be known. The program itself consists of actions whose deterministic semantics define the single operations (assignment, expression evaluation, condition evaluation, etc.) and the control flow of actions in this transformation. The control-flow semantics describe the starting point and the end points of the program as well as the transitions and the transition conditions between actions. The control flow semantics of an action define zero, one, or two alternative successors of the action. We call actions with one successor sequential actions and actions with two alternative successors branches. Actions with no successors mark program end points.

The programs considered in this work are purely computational. They are free of any communication, synchronization, or other blocking during their execution, see the *simple-task* model described in (Kopetz, 1997). Instead we assume that inputs to a program are available before its execution starts (as part of the initial store) and results are written to memory locations that are read by the I/O subsystem when the execution has completed. Also, the values of variables only change as the result of the operations performed by the program execution. There are no volatile or shared variables that change their values asynchronously to program execution.

We define an *execution path* as a sequence of actions that starts with a valid initial store at the starting point, obeys the semantics of the actions, and terminates at an end point of the program. The program code and the possible initial stores characterize the feasible execution paths of a program.

For each pair of different execution paths there is a maximal sequence of actions that is a prefix of both paths. By definition the last operation of such a prefix is a branch. As all operations are assumed to be deterministic and the actions preceding the branch are identical for both paths, the choice of different successors has to be due to differences in the initial store (i.e., input variables) of the executions. We therefore call such a branch an *input-data dependent branch.* There are also branches that are not input-data dependent. The latter do not occur as the last operations of a maximum common prefix of any two execution paths.

Each action of an execution path has an *execution time*, a positive integral number (e.g., number of processor cycles). We assume that the execution time of an action depends on the semantics of the operation it performs and the sequence of actions preceding the action on the execution path. The execution times of actions on a path are considered to be unaffected by actions that are not local to the path (e.g., the actions performed prior or in parallel to that path). The execution time of an action is further assumed to be independent of the store on which the action is performed, i.e., the durations of operations are assumed to be independent of the actual values of their operands and memory access times are assumed to be homogeneous for all variables. The *execution time of an execution path* is the sum of the execution times of the actions of the path.

## 3. THE SINGLE-PATH APPROACH

As mentioned before, WCET analysis is in general complex 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. The single-path approach avoids this complexity by ensuring that the code has only a single execution path. This approach uses code transformations to transform input-data dependent loops and branches. It transforms loops with input-data dependent termination conditions into loops with invariable iteration counts. Input-data dependent branches with the semantics of *if* or *case* statements and their alternatives are transformed into strictly sequential code. To be precise, the code resulting from the transformation of branches avoids data dependencies in execution times by keeping input-data dependent branching local to single operations with data-independent execution times.

The sequential code generated by the above-mentioned transformation includes so-called predicated operations, i.e., operations that realize branches within single machine instructions and have a constant, data-independent execution time. The predicated instruction used is the *conditional move instruction.* It is implemented on a number of modern processors (e.g., Motorola M-Core, Alpha, Pentium P6) and has the following general form:

```
movCC destination, source
```

The conditional move compares the condition code *CC* with the condition code register. If the result is *true* the processor copies the contents of the *source* register to the *destination* register. If the condition evaluates to *false*, the value of *destination* remains unchanged.

In the next section we show how the conditional move instruction and program transformations are used to create code whose execution time is constant and therefore fully predictable.

## 4. CONVERTING WCET-ANALYZABLE CODE INTO SINGLE-PATH CODE

Every well structured and WCET-analyzable piece of program code can be translated into code with a single execution path. (By WCET-analyzable code we understand code for which the maximum number of loop iterations for every loop is known. A WCET bound is thus computable.) The translation replaces all input-data dependent alternative statements and loops by deterministic code, i.e., code that restricts all input-data dependencies to conditional move instructions.

In the following we show how input-data dependent conditional statements and loops are translated. We use *if* statements with two alternatives to illustrate the translation of conditionals. The translation of conditionals with more than two alternatives is very similar and therefore not specifically described. Further, *goto* and *exit* statements are not considered in this paper. Any functionality can also be implemented without the latter.

## 4.1 Translation of Conditionals

Conditional branching statements conditionally change the values of one or more variables. The translation of conditional branches is straightforward. It generates sequential code with conditional move assignments for each of the conditionally changed variables. This conversion from control dependencies into data dependencies is called if-conversion (Allen et al., 1983, Park and Schlansker, 1991) and is traditionally only used to translate the bodies of innermost loops into non-branching code, see Figure 1.

```
                                    tmp1 := expr1;
if cond                             tmp2 := expr2;
then result := expr1;               test cond;
else result := expr2;               movt result, tmp1;
                                    movf result, tmp2;
```

*Figure 1.* Branching statement and corresponding sequential code generated by if-conversion.

When translating assignments in nested conditional branches that are input-data dependent, the conditions of all nested branches have to combined in the conditions of the generated conditional assignments.

To describe the translation of conditionals more formally, we assume that each branch uses the input variables $v_1',...,v_m'$ to compute the values for variables $v_1,...,v_n$. Thus, using if-conversion an *if* statement is translated into sequential code as shown in Figure 2.

| | |
|---|---|
| | $(h_1,...,h_n) := F1(v_1',...,v_m')$ |
| **if** *cond* | $(h_1',...,h_n') := F2(v_1',...,v_m')$ |
| **then** $(v_1,...,v_n) := F1(v_1',...,v_m')$ | *cond*: $(v_1,...,v_n) := (h_1,...,h_n)$ |
| **else** $(v_1,...,v_n) := F2(v_1',...,v_m')$ | **not** *cond*: $(v_1,...,v_n) := (h_1',...,h_n')$ |

*Figure 2.* General form of if-conversion.

The last two lines of the code on the right side of Figure 2 are guarded assignments with the guards being the condition of the *if* statement and its negation, respectively. We use these guarded assignments of tupels to represent a number of conditional move operations. The guard represents the condition of the respective conditional moves (see also Figure 1).

Figure 3 shows how nested *if* statements are translated. First, the condition of the current *if* and the enclosing conditions are combined. This new conditional is then translated into sequential code using if-conversion.

```
-- conditions so far: cond-old
if cond-new
then ...
else ...
```

```
if cond-old and cond-new
then ...
else ...
```

*Figure 3.* First step of the translation of a nested *if* statement.

## 4.2 Translation of Loops

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.

Second, the new conditional statement, that has been generated from the old loop condition, is transformed into a constant-time conditional assignment. As a result the entire loop executes in constant time.

Figure 4 illustrates the first step of the loop transformation. In the translated version (right), the variable $finished_x$ has been introduced to store the information if the original loop would have executed the current iteration or would already have terminated.

```
-- conditions so far: cond-old
while cond-new do max expr times
    stmts;
```

```
$finished_x$ := false;
for $i_x$ := 1 to expr do
begin
    if not cond-new
    then $finished_x$ := true;
    if cond-old and not $finished_x$
    then stmts;
end
```

*Figure 4.* Translation of a loop.

Applying the described transformation to existing real-time code may yield temporal predictability at very high cost, i.e., execution time. Thus we consider the illustration of the transformation as a demonstration of the general feasibility of our approach, rather than proposing to use the transformation for generating temporally predictable code from arbitrary real-time code. In order to produce 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.

## 5. AN EXAMPLE

The example illustrates the described transformation. It shows an implementation of *bubble sort* and the corresponding single-path code.

On its left side Figure 5 lists a typical traditional implementation *of bubble sort.* The function has one parameter, the array *a* to be sorted. The function uses two nested loops to transport elements to their correct positions. In each iteration of the inner loop two neighbouring array elements are compared. If the comparison evaluates to true the two elements are swapped, otherwise no operation is performed.

```
static void bubble1(int a[])
{
  int i, j, t;
  for(i=SIZE-1; i>0; i--)
  {
    for(j=1; j<=i; j++)
    {
      if (a[j-1] > a[j])
      {
        t = a[j];
        a[j] = a[j-1];
        a[j-1] = t;
      }
    }
  }
}
```

```
static void bubble2(int a[])
{
  int i, j, s, t;
  for(i=SIZE-1; i>0; i--)
  {
    for(j=1; j<=i; j++)
    {
        s = a[j-1];
        t = a[j];
        s <= t: a[j-1] = s;
        s > t:  a[j-1] = t;
        s <= t: a[j] = t;
        s > t:  a[j] = s;
    }
  }
}
```

*Figure 5.* Traditional (left) and single-path (right) version of *bubble sort.*

Note that the branching statement causes an execution-time variability in the inner-loop body. Depending on the result of the branching condition the duration of each iteration of the inner loop is either long or very short. As the body of the inner loop executes *SIZE(SIZE-1)/2* times when sorting an array of *SIZE* elements, one immediately concludes that *bubble1* has at least *SIZE(SIZE-1)/2+1* possible execution times. If branch prediction hardware is used the number of possible execution times is in fact much higher.

On its right side Figure 5 shows the *bubble sort* code after the transformation of input-data dependent branches. The *if* statement in the inner loop has been replaced by four conditional move instructions.

To compare the execution characteristics of the two implementations we generated executable programs for both versions. The *bubble1* version was directly compiled and linked for the Motorola M-Core processor (Motorola, 1997). As our compiler does not translate the guarded assignments we produced the code for *bubble2* by editing the machine code of *bubble1* – we replaced the conditional branch of the *if* statement by sequential code, including conditional move instructions. Both versions were then tested on a cycle-accurate M-Core simulator. The results are summarized in Table 1. For the experiment an array size of 10 was assumed.

*Table 1.* Number of execution paths, minimum and worst-case execution time of *bubble sort* variants for array size 10.

| *Implementation* | *# Paths* | *min. Exec. Time* | *WCET* |
|---|---|---|---|
| Traditional | 3628800 | 675 | 810 |
| Single Path | 1 | 972 | 972 |

The traditional version of *bubble sort* has a big variability of execution times (675-810 CPU cycles). As expected, the single-path version has a single execution time for all inputs (972 cycles). The WCET of the single-path implementation is about 20% greater than that of the conventional implementation. This seems to be a reasonable price given the fact that finding this execution time is trivial – there is only one path to evaluate. The traditional solution has more than 3.6 million paths (a huge number given that simple piece of code). While identifying the worst-case path is not too difficult for this simple, one can immediately think of more complex code with a much greater number of paths that are not so easy to analyze. In that case the advantage of having a single path is obvious. As path analysis is unnecessary the problems of path analysis, i.e., potential flaws and pessimism, are non-existent.

## 6. CONCLUSION

In an earlier paper we presented the single-path programming paradigm. This programming paradigm makes it possible to write programs that can be easily analyzed for their WCET. The key element of this programming paradigm is to keep input-data dependent branching local to single machine instructions with invariable execution times. The conditional move instruction is the most important of those instructions.

In this paper we showed how every piece of code that is WCET-analyzable (i.e., the maximum number of iterations can be bounded for all loops) can be transformed into single-path code. This is done by converting all input-data dependent conditional statements and loop statements and by using if-conversion to translate all remaining input-data dependent branches into sequential code with conditional moves. WCET analysis for the single-path code is trivial: The WCET is obtained by executing the code with any valid input data on a hardware simulator or even the target hardware and measuring the execution time of the single execution path.

Although we demonstrated how input-data dependent code is transformed we do not consider this transformation as an adequate method for producing single-path code from any piece of code – Relying purely on the transformation will, in general, leave the programmer with very inefficient code. To get code that cannot only easily be analyzed for its WCET but also has a good performance, developing or selecting algorithms where execution paths do only marginally or not at all depend on input data is important. Identifying and developing algorithms that are well-suited for our approach will be a central focus of our further research.

## REFERENCES

Allen, J., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of Control Dependence to Data Dependence. *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*: 177-189.

Colin, A., and Puaut, I. 2000. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 18(2/3):249-274.

Engblom, J., and Ermedahl, A. 2000. Modeling Complex Flows for Worst-Case Execution Time Analysis. *Proceedings of the 21st IEEE Real-Time Systems Symposium*: 163-174.

Kopetz, H. 1997. *Real-Time Systems*. Kluwer Academic Publishers.

Lundqvist, T., and Stenström, P. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. *Proceedings of the 20th IEEE Real-Time Systems Symposium*: 12-21.

Motorola Inc. 1997. *M-Core Reference Manual.*

Park, J., and Schlansker, M. 1991. On Predicated Execution. *Technical Report HPL-91-58*, Hewlett Packard Software and Systems Laboratory, Palo Alto, CA, USA.

Petters, S., and Färber, G. 1999. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. *Proceedings of the 6th IEEE International Conference on Real-Time Computer Systems and Applications*: 442-449.

Puschner, P., and Burns, A. 2000. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115-127.

Puschner, P., and Burns, A. 2002. Writing Temporally Predictable Code. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.*