Writing Temporally Predictable Code [*]

Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
A1040 Wien, Austria
Email: peter@vmars.tuwien.ac.at

Alan Burns
Department of Computer Science
University of York
York, YO10 5DD, United Kingdom
Email: burns@cs.york.ac.uk

Abstract

The Worst-Case Execution-Time Analysis (WCET Analysis) of program code that is to be executed on modern processors is a highly complex task. First, it involves path analysis, to identify and describe the possible execution paths through the code. Second, it models the worst-case timing of possible paths on the target hardware, where the characterization of the timing of sophisticated hardware features (e.g., instruction pipelines, caches, parallel execution units) and their interferences are non-trivial.

This paper presents a programming paradigm that takes the complexity from WCET analysis. Program code written according to this paradigm only has a single execution path. Writing single-path code makes path analysis and thus WCET analysis trivial. The WCET of the single path is obtained by executing the code (necessarily on that single path) and logging the duration of this execution. To demonstrate that the single-path approach provides a universal solution to the WCET-analysis problem, the paper shows how every WCET-analyzable piece of code can be translated into single-path code.

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) WCET analysis of real-time tasks and many sub-problems of WCET analysis have been solved [9].

[*]This work has been supported in part by the IST research project "Systems Engineering for Time-Triggered Architectures (SETTA)" under contract IST-10043.

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 for the following reasons: 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 [2, 3]. 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 instruction-execution units, and branch-prediction to enhance execution performance. These mechanisms are complex in their implementation and have mutual interferences in their timing. Besides that, the mechanisms and their timing are generally scarcely documented to protect the manufacturer's intellectual property [8]. These facts taken together make it difficult if not impossible, to build tools for static WCET analysis of modern processors that can be guaranteed to model the timing of the processor correctly.

3. *Complexity of analysis*: Besides the mentioned difficulties in identifying the possible execution paths and obtaining hardware-timing data, the complexity of WCET analysis itself is a problem. 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 [5]. 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 this paper we present a radical programming paradigm that avoids the described problems. This paradigm yields completely temporally predictable programs and allows for a simple WCET analysis. The central idea of the presented paradigm is that programmers write programs whose behaviour is independent of input data and which thus always execute on one single execution path.

The fact that programs only have a single execution path in fact makes WCET analysis trivial: First, path analysis is superfluous – observing the execution path of any code execution with any input data yields the singleton execution path. Second, there is no need for building and using complex and accurate hardware timing models for static WCET analysis. A central reason for performing static WCET analysis with such complex models was that a measurement-based WCET analysis was infeasible. Due to the high number of execution paths exhaustive tests were impossible and one could not be sure that the measured scenarios indeed covered the worst case. In contrast, if programs only have a single path, as in the proposed approach, this singleton path is necessarily the worst-case path. Thus, obtaining the WCET by "exhaustive" measurements is possible (either by measurements on the target or on a cycle-accurate hardware simulator) and there is no need to build any specific tools for static analysis.

An approach that only permits programs with a single execution path may seem to be very restrictive and to allow programmers to write only very simple programs. It will be shown later that the proposed approach is, however, not at all that restrictive. In fact, any piece of code that is WCET-analyzable can be translated into code with a single execution path. The translation uses if-conversion [1] to produce code that keeps input-data dependent alternatives in the code local to single conditional operations with data-independent execution times.

The paper is structured as follows: Section 2 explains the terms and assumptions used throughout the paper. Section 3 introduces the single-path approach. It describes the programming paradigm and shows how WCET-analyzable programs are transformed into programs with a single execution path. Section 4 provides a program example that demonstrates the single-path approach and illustrates some of its properties. Section 5 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 defines 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 [4]). 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 each such prefix is a branch. As all operations are assumed to be deterministic and the actions preceeding 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, the problem of 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. 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.

3.1 Constant-Time Conditional Expression

The key feature to our predictable-programming approach is the so-called *constant-time conditional expression* operation. A constant-time conditional expression 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 constant-time conditional expressions.

```
Cond # Expr1 : Expr2
```

Cond represents the condition of the constant-time conditional, *Expr1* and *Expr2* stand for the two expressions that are evaluated. If *Cond* yields *true* then the value of the constant-time expression is the result of *Expr1*. If *Cond* evaluates to *false* the constant-time expressions returns the value of *Expr2*.

What has been described above may remind the reader of the conditional assignment operator "?:" of the C programming language. Indeed, as we assume that *Expr1* and *Expr2* 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 constant-time conditional 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 constant-time conditional 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. The different semantics of the two conditional operators are illustrated in Figure 1.

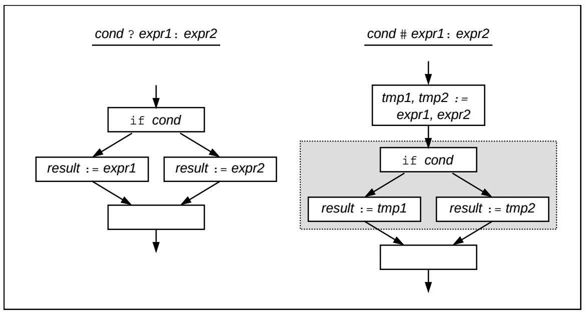

Figure 1. Semantics of the standard C conditional operator (left) and the constant-time conditional operator (right).

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 constant-time conditional 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, see gray box in Figure 1. The realization of this operation is discussed in the following section.

3.2 Conditional Move Instruction

The implementation of the selection and the assignment of the final result of the constant-time conditional is not as straightforward as it might look at the first sight. Although the two operations of the two alternatives are equal, implementing the two assignments as two alternatives in a conditional branching statement may yield different execution times depending on which alternative will actually be executed. There are the following possible reasons for this:

1. The time needed for the branching itself differs for the two alternatives. E.g., in a pipelined CPU the pipeline may stall on a jump to the second alternative while no stall occurs if the first alternative is executed.

2. In computer systems with instruction cache, executing either of the two alternative instructions of a branch in general leaves the cache in different states. Different cache states potentially influence the execution times of instructions further downstream the execution in different ways, thus making an exact execution-time prediction impossible. The fact that a program may execute a large number of branches with unpredictable timing leads us back to the problem outlined in Section 1, Item 3.

Note: The first problem is, in principle, solvable by stuffing the shorter alternative with additional functionalily or NOP instructions until both alternatives have equal execution times (Adjusting branch alternatives to complete in equal time might actually look like a general solution to achieving predictable execution times). In practice, however, the implementation of this technique requires detailed knowledge about the operation and the timing of the target system, which is in general not available (see Section 1, Item 2). Also, the second problem remains.

The central deficiency of the "solution" considered so far is that it still does not yield a unique execution time – still different input data cause programs to execute different execution paths, which might lead to variable timing. The real solution we propose totally avoids input-data dependent branches at the instruction level. This solution is generated by using if-conversion [1, 7] to translate input-data dependent branches into sequential code. The generated sequential code includes so-called predicated 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, SPARC V9). The general conditional move instruction has the following form:

```
movCC destination, source
```

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

The Motorola M-Core processor [6] supports the conditional move instructions `movt` (move conditionally if *CC* is true) and `movf` (move conditionally if *CC* is false). Using these instructions the constant-time conditional assignment can be used to implement the conditional operator as shown in the following piece of pseudo machine code:

```
r1 = expr1;
r2 = expr2;
test cond;
movt rr, r1;
movf rr, r2;
```

First the two expressions *expr1* and *expr2* are evaluated. The results are stored in registers *r1* and *r2*, respectively. Then the condition *cond* is evaluated. Depending on the result of the condition evaluation either of the predicated operations `movt` – in case the condition evaluates to true – or `movf` – in case the condition yields false – assigns the new value, *r1* or *r2*, to register *rr*. At the end of this instruction sequence *rr* holds the correct result of the conditional assignment.

This implementation of the conditional assignment has no branches. It thus facilitates the implemenation of any conditional code in a strictly sequential manner. The resulting code has a single, hence predictable execution time. Note that this type of conditional assignment also works if more than one variable needs to be assigned different values depending on a condition. In that case code for computing intermediate results and appropriate conditional move instructions need to be generated for each of those variables.

3.3 Converting WCET-Analyzable Code into Single-Path Code

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 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.) This translation replaces all input-data dependent branches by sequential code that uses constant-time conditional expressions.

In high-level languages data dependent branching occurs in conditional statements (e.g., if statements) and loops – we do not consider gotos and exit statements here. 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.

4 An Example

The example demonstrates how the non-jumping conditional assignment is used to write code with predictable execution time. The example shows two different implementations of *bubble sort*.

Figure 2 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.

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

```
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])
      {
          /* swap */
          t = a[j];
          a[j] = a[j-1];
          a[j-1] = t;
      }
    }
  }
}
```

Figure 2. Traditional implementation of *bubble sort*.

body of the inner loop executes $\frac{SIZE\times(SIZE-1)}{2}$ times when sorting an array of *SIZE* elements, one immediately concludes that *bubble1* has at least $\frac{SIZE\times(SIZE-1)}{2}+1$ possible execution times. In the presence of branch prediction mechanisms the number of possible execution times is in fact much higher.

Figure 3 presents an alternative implementation of bubble sort. This implementation is free of conditional branches. Instead it uses two non-jumping conditional assignment operations.

```
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];
        a[j-1] = ((s<=t) # s : t);
        a[j] = ((s>t) # s : t);
    }
  }
}
```

Figure 3. Single-path implementation of *bubble sort*.

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. As we do not (yet) have a compiler that translates the "#:" operator we produced the code for *bubble2* by editing the machine code of *bubble1* – we replaced the conditional branch of the if statement by strictly sequential code, including conditional move instructions. Both versions were then tested on a cycle-accurate M-Core simulator. The characteristics of the two implementations 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.ET*	*WCET*
Traditional	3628800	675	810
Single Path	1	972	972

The traditional version of *bubble sort* is characterized by a big variability of execution times (675–810 CPU cycles). As expected, the single-path version has a single execution time for all possible inputs (972 cycles). The WCET of the single-path implementation is about 20% greater than the WCET 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 and small piece of code). While identifying the worst-case path is not too difficult for this simple algorithm from the well-studied problem domain of sorting, 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.

The used example is rather simple and, as mentioned above, the WCET penalty of the single-path solution is reasonable considering the benefits of the solution. Providing a more generalized evaluation of the single-path programming paradigm with respect to the achievable worst-case performance is difficult as it is easy to construct examples which have a very extended execution time. However real code developed for embedded real-time applications is often quite constrained and a 20% increase in execution time over traditional programming (as illustrated by the example) is perhaps typical.

5 Conclusion

The paper 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 the constant-time conditional expression. This expression implements a logical branch (e.g., if statement) as strictly sequential code of constant execution time. This is achieved by keeping the logical branch local to a single machine instruction with invariable execution time, the conditional move instruction.

It has been shown that every piece of code that is WCET-analyzable (i.e., the maximum number of iterations can be bounded for all loops of that code) can be transformed into single-path code. This is done by using if-conversion and translating all input-data dependent branches into constant-time conditional expressions. Having performed the transformation, 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.

The demonstration that any piece of WCET-analyzable code can be transformed into single-path code should be considered rather as a proof of the generality of the concept than a general method for producing single-path code. In fact, 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.

Acknowledgments

We would like to thank Raimund Kirner for his valuable comments on an earlier version of the paper and Bernhard Scholz for pointing us to related work on if-conversion.

References

[1] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, Texas, USA, Jan. 1983.

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

[3] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174, Orlando, Florida, USA, Dec. 2000.

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

[5] T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, Phoenix, Arizona, USA, Dec. 1999.

[6] Motorola Inc. *M-Core Reference Manual*, 1997.

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

[8] S. Petters and G. Färber. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proceedings of the 6th IEEE International Conference on Real-Time Computer Systems and Applications*, pages 442–449, Hong Kong, China, Dec. 1999.

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