DETC2007-35375

TOWARDS A GENERIC “SINGLE-PATH PROGRAMMING” SOLUTION WITH REDUCED POWER CONSUMPTION

Ayman K. Gendy

Embedded Systems Laboratory

Michael J. Pont*

Embedded Systems Laboratory

University of Leicester, University Road,
Leicester LE1 7RH, UK.

ABSTRACT

As embedded designs become more widespread and complex they tend to use more modern processors. Such processors will often include features (such as pipelines, caches, and branch predictors) which help to improve performance. While such performance improvements are welcome, they come at the price of predictability. More specifically, the use of advanced processor hardware makes it difficult to predict the worst-case execution time (WCET) of tasks. As part of an effort to address these problems, Puschner and Burns (Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Jan. 2002) proposed the “single path programming paradigm”. As its name implies program code written according to this paradigm has only one execution path: this helps to ensure a constant execution time. Yet there are two problems with the techniques described by Puschner and Burns: (i) they are applicable only to hardware which supports “conditional move” or similar instructions; (ii) their balancing approach increases power consumption. In the present paper, we begin to address both of these problems with a set of novel code-balancing techniques. The effectiveness of these new techniques is explored by means of an empirical study.

1 INTRODUCTION

Information about the upper bound of task execution time is a key factor when designing real-time embedded systems. This “worst case execution time” (WCET) is defined as the longest time taken by the processor to execute a task in the absence of pre-emption [1]. WCET estimates can be used as the basis of many key design operations, such as scheduling and schedulability analysis, determining whether performance goals are met for periodic tasks, checking that interrupts have sufficiently short reaction times, and finding performance bottlenecks [2].

Determining WCET values is becoming more challenging as embedded designs become more complex and make use of faster and smaller processors and “system on chip” architectures [3],[4]. These modern processors often incorporate features (such as pipelines, caches, and branch predictors) that help to increase the performance. Unfortunately, these features also make it difficult to determine the internal state of the system [5]. As a consequence, analysing the timing of the whole system and estimating the WCET requires a significant effort even for systems with a simple software architecture, such as time-triggered systems which use a table-driven task schedule [6]. In addition, use of fault-tolerance policies based on knowledge of WCET becomes more challenging with modern processors [7].

To cope with these difficulties some researchers base their scheduler on expected execution time rather than WCET [8], [9]. This represents - at best - a partial solution where precise timing behaviour is required.

A number of previous studies have been conducted to address the problem of accurately estimating WCET: this work has involved analysis and / or measurements (e.g. [2], [5], [10], [11]).

For example, Engblom and Jonsson [2] examined the timing of instructions from the perspective of static WCET using a mathematical model of instruction execution on in-order single-issue pipelined processors: the particular concern in this study was with timing effects between non-adjacent instructions. Engblom and Jonsson showed that there are negative long timing effects (LTEs) which can be safely ignored, and positive LTEs which have to be accounted for in a WCET analysis. In another study, Rochange and Sainrat [10] showed that pure static analysis might not allow safe WCET computation for modern processors with speculative execution. In particular, they noted that when a

* Author of correspondence, Phone: +44 (0)116 252 5052, Fax: +44 (0)116 252, Email: M.Pont@le.ac.uk.

branch instruction is predicted, the instructions belonging to the predicted path can be executed before that the branch is resolved. They discuss the possible effects of executing the wrong path whenever a branch is mispredicted.

Deverge and Puaut [5] explore the issues to be addressed for designing a measurement-based method for WCET estimation. They propose generation of test data for program segments, using program clustering, to combine execution time of program segments and to obtain the WCET of the whole program.

A different approach has been proposed by Puschner and Burns [11]: this is called “the single-path programming paradigm”. This approach was based on the idea of writing the program in manner which ensures that there is only one execution path. They showed that this helps to produce a constant execution time.

In this paper, we highlight two issues with the techniques described by Puschner and Burns: (i) they are applicable only to hardware which supports “conditional move” or similar instructions; (ii) their balancing approach can increase power consumption. In the present paper, we address both of these problems with a modified set of single-path programming techniques. The effectiveness of these new techniques is demonstrated by means of an empirical study.

The remainder of this paper is organised as follows. Section 2 gives an overview of the “single path programming paradigm” as introduced by Puschner and Burns [11]. In Section 3 we introduce some code-balancing techniques which address two issues with this paradigm. In Section 4 we assess the proposed code-balancing techniques. In Section 5 we present our conclusions.

2. THE SINGLE PATH PROGRAMMING PARADIGM

This section gives an overview of the single-path programming paradigm as described previously ([11] - [14]).

Programming code that complies with the single-path programming paradigm has only one execution path. This can be achieved by replacing input-data dependencies in the control flow by predicated (instead of branched) code. In predicated execution, instructions are associated with predicates: if the predicate evaluates to true the instruction executed; otherwise the microprocessor internally replaces the instruction by a no-operation (NOP) instruction. It is assumed that a simple predicated execution model is used (such as the conditional move instruction in M-Core processor, in which conditional instructions have a constant, data-independent execution time).

As an example, Figure 1 shows some pseudo code that indicates how a code branch using if-then-else structure can be translated to the single path form. In this example the conditional move instruction “movt” copies the value of “temp1” to “result” if the result of the “test” instruction is true; otherwise the processor performs a NOP instruction. The same can be said for the “movf” instruction; it will copy the value of “temp2” to “result” if the result of the “test” instruction is false; otherwise the processor performs a NOP

instruction. This code can be easily modified to be used with a nested if statements [15].

Figure 1. Converting if-then-else conversion to single path (adapted from [14]).

In a similar manner, a loop of variable length can be translated into a loop of constant length (provided we know the maximum size of the loop). Please note that less structured “goto” and “exit” statements are not considered in this approach.

It has been demonstrated ([11]-[14]) that using this method helps to produce a constant execution time. However, this method has some drawbacks:

- Its usage is limited to hardware which supports “conditional move” or similar instructions
- It is likely to increase power consumption because the CPU will always execute the single-path code for a fixed (maximum) period. During this time, the processor will be in “full power” mode.

3. PROPOSED CB1 TECHNIQUES

In this section we begin to address the drawbacks mentioned above by using a set of novel code-balancing techniques. For ease of reference, we refer to the approach described here as the “CB1 techniques” in the remainder of this paper.

3.1 Overview

The main idea behind the CB1 approach can be explained by considering an example. This example is intended to stabilise the time taken to complete a number of iterations in a given loop.

Assume that the time spent in performing “ x ” iterations of the loop is equal to $\text{Time}(x)$, where: $(1 \leq x \leq \text{MAX})$ and MAX is the maximum number of iterations. The microcontroller is set to enter a power-saving mode for the period of time required to perform $(\text{MAX} - x)$ iterations. This time can be approximated by Eq. (1).

$$\text{Time}(\text{MAX} - x) = \frac{(\text{MAX} - x) \times \text{Time}(x)}{x} \quad (1)$$

Hardware timers can be used to measure $\text{Time}(x)$; time spent in performing x iterations, by starting the timer directly before the start of the loop and stop it directly after the last

statement of the loop. By substituting this value in Eq. (1), Time (MAX - x) can be calculated. This time can then be used to set a timer interrupt to waken the microcontroller after the required time has elapsed.

Please note that it is assumed that the loop will be executed at least once for Eq. (1) to give real results. Also note that there is an approximation in calculating Time (MAX - x) given by Eq. (1) as it does not take into account the effects of the performance improvements features mentioned earlier. The reason for this approximation is to simplify the calculations and the implementations process so as to be suitable for any platform while avoiding complex analysis of each specific feature.

Based on this form of “sandwich delay”, a set of balanced code can be used to reduce the variations in executing for and while loops. The approach can also be used to balance if-then-else structures, as explained in the following subsections.

3.2 Balanced for loop

Listing 1 shows pseudo code that can be used to stabilise the WCET of a *for*-loop for any number of iterations in the range of [1, MAX], where MAX is the maximum number of iterations.

Please note that a small “safety margin” was added to the time calculated in Eq. (1) to assure that there is time to enter sleep mode before the interrupt occurs even at the maximum loop length:

```
Start timer;

for (i = 0; i < x; i++)
{
    //loop body
}

Stop timer;
Time(x) = timer count;
Time(MAX - x) = (MAX - x) X Time(x)/ x;
Reset timer;
Adjust timer interrupt to occur after time:
    Time (MAX - x) + “Safety margin”;
Send the microcontroller to power saving mode;
```

Listing 1. Pseudo code of a balanced *for*-loop.

3.3. Balanced while loop used for waiting for input

Listing 2 shows a pseudo code that can be used to stabilise the WCET of executing a while-loop which is usually used to wait, for a predefined maximum time (TMAX), for an input to be ready.

Please note that a small “safety margin” was (again) added to the time TMAX after the end of the while loop to ensure that there is time to enter sleep mode before the interrupt occurs, even in case where the input becomes ready at time TMAX. The safety margin will typically be 1% of the value of TMAX.

Set timer interrupt to occur after Time = TMAX,
(where TMAX equals to maximum time of waiting for
input to be available)

```
Start timer;

while (1)
{
    if input is available or timer
        count equals to TMAX then
        break;
}
Stop timer;
Adjust timer interrupt to occur after time:
    (TMAX + “safety margin” - “timer value”);
Send the microcontroller to power saving mode;
```

Listing 2. Pseudo code of a balanced *while*-loop
used for waiting for input.

3.4. Balanced if-then-else structure

Listing 3 shows a pseudo code that can be used to stabilise the WCET of a general *if-then-else* structure.

Please note that the number of the assignment instructions in the if-part must be equal to those in the else-part (“NOP” padding or similar approaches must be used, if necessary).

```
temp1 = expr1;
temp2 = expr2;

if (cond)
{
    result = temp1;
}
else
{
    result = temp2;
}
```

Listing 3. Pseudo code for a balanced *if-then-else*
structure (adapted from [14]).

4. PERFORMANCE OF THE CB1 TECHNIQUES

An empirical test was carried out to explore the effectiveness of the CB1 techniques. The procedure and the results obtained are detailed in this section.

4.1. Initial test

In this test an example which was used to assess the effectiveness of the single path programming paradigm (in [11]) is used here. The original example explores different implementations of a “bubble sort” for arrays of 10 elements. The original example was used to sort all the elements of the array. The version used here used here sorts the first x elements of the array (where x is $\leq$ SIZE, the total number of the array elements). This modification was made in order to explore the impact of different implementations on the execution time, jitter, and power consumption.

Our tests employed using a time triggered co-operative (TTC) scheduler [16]. The tick interval was set to 10 ms.

The main (sorting) task was run every two ticks.

Three additional tasks were also scheduled:

- i) A “jitter-test” task. This low-priority task was scheduled to execute in the same tick as the sorting task. It was used to measure the effect of the variations in the execution time of the sorting task on the jitter in the start times of other tasks in the system
- ii) A “sort length” task used to increment the value of x , from 2 to 10 and then back to 2.
- iii) A “sort complexity” task used to initialise the array in “completely sorted” or “completely unsorted” forms, in order to vary the time taken to carry out the sort process.

The test was carried out on an NXP (formerly Philips) LPC2106 microcontroller running on a small evaluation board. The LPC2106 is based on an ARM7TDMI core and is typical of modern (low cost) embedded processors. Because this microcontroller does not support the conditional move instruction the single path code is modified, while keeping the main structure described by [11], to cope with this limitation. Listing 4, Listing 5, and Listing 6 show different implementations of the bubble sort using the traditional, single path, and the proposed CB1 code respectively.

Table 1 and Table 2 show the measured minimum and maximum execution time, maximum jitter, and average power consumption resulted from each implementation. From these tables it can be noticed that:

- Both the single-path code and CB1 code demonstrated a reduction in both the variation of the execution time and in jitter levels. These improvements were at the expense of an increase in the average power consumption and execution time.
- The jitter and the variations in execution time obtained by using the CB1 code was less than that of the traditional code and higher than that of the single-path code.
- The average power consumption obtained by using the CB1 code was less than that of the single path code and higher than that of the traditional code.

```
void traditional_bubble( )
{
    int i,j,t;
    for (i=x-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;
            }
        }
    }
}
```

Listing 4. Traditional implementation of bubble sort.

```
void single_path_bubble( )
{
    int i,j,s,t, dummy[10];
    char Finished_i,Finished_j;

    for (i=SIZE-1; i>0; i--)
    {
        if (i > (x-1))
        { Finished_i = 1; }

        if (!(i > (x-1)))
        { Finished_i = 0; }

        for (j=1;j<= SIZE-1;j++)
        {
            if (!(j <= i))
            { Finished_j = 1; }

            if (j <= i)
            { Finished_j = 0; }

            if (!Finished_i && !Finished_j)
            { //do real statements
                s = a[j-1];
                t = a[j];

                if (s <= t)
                {
                    a[j-1] = s;
                    a[j] = t;
                }

                if (s > t)
                {
                    a[j-1] = t;
                    a[j] = s;
                }
            }

            if (!(Finished_i &&
                !Finished_j))
            { //do dummy statements
                s = dummy[j-1];
                t = dummy[j];

                if (s <= t)
                {
                    dummy[j-1] = s;
                    dummy[j] = t;
                }

                if (s > t)
                {
                    dummy[j-1] = t;
                    dummy[j] = s;
                }
            }
        }
    }
}
```

Listing 5. Single path implementation of bubble sort (adapted to work without the support of the conditional move instruction).

```
void balanced_code_bubble()
{
    int i,j,s,t;
    unsigned long Temp1=0,Temp2=0;
    // Initialise and start T1
    T1MCR = 0x00;
    T1TCR = 0x2;    // Reset counter
    T1TCR = 0x01;    // Counter enable
```

```

for (i=x-1;i>0;i--)
{
//Store T1 value before inner loop
T1TCR = 0x00;
Temp1 = Temp1 + T1TC;

// Prepare T1 for inner loop
T1MCR = 0x00;
T1TCR = 0x2;
T1TCR = 0x01;

for (j=1;j<=i;j++)
{
s = a[j-1];
t = a[j];
if (s<=t)
{
a[j-1] = s;
a[j] = t;
}
if (s > t)
{
a[j-1] = t;
a[j] = s;
}
}
// Store T1 value at end of inner
// loop
T1TCR = 0x00;
Temp2 = T1TC;

// Adjust MR for the remaining
// time,
// start T1 and go to sleep
T1MR0 = (( SIZE- i)*Temp2 )/i;
T1TCR = 0x2;
T1MCR = 0x05;
T1TCR = 0x01;
PCON = 1;      // Go to sleep

// Complete the outer loop
// Initialise and start T1
T1MCR = 0x00;
// Add the time spent in inner
// loop to the time spent so
// far in the outer loop
Temp1 = Temp1 + Temp2 + T1TC;
T1TCR = 0x2; // Reset counter
T1TCR = 0x01; // Counter enable
}

// Stop T1 and calculate the remaining
// time
// in the outer loop
T1TCR = 0x00; // Stop counter
Temp1 = Temp1 + T1TC;
// Adjust MR for the remaining time,
// start T1 and go to sleep
T1MR0 = (( SIZE- (x-1))*Temp1)/(x-1);
T1TCR = 0x2; // Reset counter
T1MCR = 0x05;
T1TCR = 0x01; // Counter enable
PCON = 1;      // Go to sleep
}

```

Listing 6. CB1 implementation of bubble sort

Table 1. Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of bubble sort.

	<i>Min (ms)</i>	<i>Max (ms)</i>	<i>(Max- Min) (ms)</i>	<i>% of variations</i>
Traditional	0.00815	0.33165	0.32350	97.54 %
CB1	0.90050	1.00265	0.10215	10.19 %
Single-path	0.80210	0.85305	0.05095	5.97 %

Table 2. Maximum jitter and average power consumption resulted from different implementations of bubble sort.

	<i>Maximum jitter (ms)</i>	<i>Average power consumption (mw)</i>
Traditional	0.3208	11.8120
CB1	0.0689	13.0185
Single path	0.0515	13.0193

4.2. Extended test

The experiment described in the previous section was repeated using two additional benchmark test cases which have been used in previous WCET studies [17], [18]:

- The first test case implements a single nested loop used to calculate the Fibonacci series for up to 30 elements.
- The second test case implements a triple-nested loop used to calculate Matrix multiplication of two 2-D arrays up to 20x20 elements in size.

The length of the tick interval of the TTC scheduler was set to 10 ms for the first test and 100 ms for the second test.

Table 3 through Table 6 show the measured minimum and maximum execution time, maximum jitter, and average power consumption resulted from each case. These results were in line with the results obtained from the test described in the previous section.

Table 3. Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of Fibonacci.

	<i>Min (ms)</i>	<i>Max (ms)</i>	<i>(Max- Min) (ms)</i>	<i>% of variations</i>
Traditional	0.00565	0.08810	0.08245	93.59 %
CB1	0.11345	0.17700	0.06355	35.90 %
Single path	0.16640	0.16635	0.00005	0.03 %

Table 4. Maximum jitter and average power consumption resulted from different implementations of Fibonacci.

	<i>Maximum jitter (ms)</i>	<i>Average power consumption (mw)</i>
Traditional	0.0829	11.2170
CB1	0.0631	11.2684
Single path	0.0005	11.4732

Table 5. Minimum, maximum, (maximum – minimum), and percentage of variations (w.r.t. the maximum) in task execution time resulted from different implementations of Matrix Multiplication.

	<i>Min (ms)</i>	<i>Max (ms)</i>	<i>(Max- Min) (ms)</i>	<i>% of variations</i>
Traditional	0.06760	41.62990	41.56230	99.84 %
CB1	52.39640	59.85655	7.46015	12.46 %
Single path	66.25970	66.49350	0.23380	0.35 %

Table 6. Maximum jitter and average power consumption resulted from different implementations of Matrix Multiplication.

	<i>Maximum jitter (ms)</i>	<i>Average power consumption (mw)</i>
Traditional	41.5651	15.8626
CB1	3.5232	18.2561
Single path	0.2366	39.0217

5. CONCLUDING REMARKS

The CB1 techniques introduced in this paper involved two stages:

- using an interrupt-based sandwich delay to keep the execution time of tasks fixed without requiring significant increases in system power consumption.
- calculating the max execution time (and required timer settings) for each form of branch / loop structure.

It has been demonstrated (using empirical studies) that:

- The variation in task execution time obtained by using the CB1 code was less than that of the traditional code and higher than that of the single-path code.
- As a consequence of the above, the task jitter levels obtained by using the CB1 code were also less than that of the traditional code and higher than that of the single-path code.
- The average power consumption obtained by using the CB1 code was less than that of the single-path code and higher than that of the traditional code.
- These single-path and CB1 results were achieved at the expense of an increase in the maximum task execution time.

Further work will be carried out to extend the CB1 techniques.

6. ACKNOWLEDGMENTS

The authors would like to thank Kam L. Chan for his help in making the power measurements.

REFERENCES

- [1] Buttazzo, G. C., 2005, "Rate monotonic vs. EDF: Judgement day", *Real-Time Systems*, 29(1), pp. 5-26.
- [2] Engblom, J. and Jonsson, B., 2002, "Processor pipelines and their properties for static WCET analysis", In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, London, UK, Springer-Verlag., pp. 334 -348.
- [3] Baynes, K., Collins, C., Fiterman, E., Ganesh, B., Kohout, P., Smit, C., Zhang, T., and Jacob, B., "2003, "The performance and energy consumption of embedded real-time operating systems", *IEEE Transactions on Computers*, 52(11), pp. 1454- 1469.
- [4] Kirner, R. and Puschner, P., 2003, "Discussion of Misconceptions about Worst-Case Execution-Time Analysis", 3rd Euromicro International Workshop on WCET Analysis.
- [5] Deverge, J. and Puaut, I., 2005, "Safe Measurement-based WCET Estimation", in: *Proc. of the 5th Workshop on Worst-Case Execution Time Analysis*, held in conjunction with the 17th Euromicro Conference on Real-Time Systems, July 2005, pp. 7-10.
- [6] Puschner, P. and Kirner, R., 2006, "From Time-Triggered to Time-Deterministic Real-Time Systems", 5th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006).
- [7] Nett, E., H. Streich, et al., 1996, "Adaptive Software Fault Tolerance Policies with Dynamic Real-Time Guarantees", *WORDS 96, IEEE Second Int. Workshop on Object-oriented Real-time Dependable Systems*, Laguna Beach, California, U.S.A.
- [8] Becker, L. B., Nett, E., Schemmer, S., Gergeleit, M., 2003, "Robust scheduling in team-robotics", 11th International Workshop on Parallel and Distributed Real-Time Systems, Nice, France.
- [9] Becker, L. B. and Gergeleit, M., 2001, "Execution Environment for Dynamically Scheduling Real-Time Tasks", *RTSS 2001, 22nd IEEE Real-Time Systems Symposium*. London.
- [10] Rochange, C. and Sainrat, P. (2002). Difficulties in computing the WCET for Processors with Speculative Execution, 2nd Intl. Workshop on Worst Case Execution Time Analysis, Vienna, June 2002.
- [11] Puschner, P. and Burns, A., 2002, "Writing Temporally Predictable Code", In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2002, pp. 85–91.
- [12] Puschner, P., 2002, "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*, Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002), pp. 163–172.
- [13] Puschner, P., 2002, "Is WCET Analysis a Non-Problem? Towards New Software and Hardware Architectures", 2nd Intl. Workshop on Worst Case Execution Time Analysis, Vienna, Austria.

- [14] Puschner, P., 2003, "The Single-Path Approach Towards WCET-Analysable Software", In Proc. IEEE International Conference on Industrial Technology, , Dec. 2003, pp. 699–704.
- [15] Allen, J., Kennedy, K., Porterfield, C., and Warren, J., 1983, "Conversion of Control Dependence to Data Dependence," In Proc. 10th ACM Symposium on Principles of Programming Languages, pp. 177–189.
- [16] Pont, M. J., 2001, "Patterns for time-triggered embedded systems", Addison-Wesley
- [17] Engblom, J., and A. Ermedahl, 2000, "Validating a Worst-Case Execution-Time Analysis Method for an Embedded Processor", In Proc. 21st IEEE Real-time Systems Symposium (RTSS'00).
- [18] Engblom, J., A. Ermedahl, et al., 2001, "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems", Journal of Software Tools for Technology Transfer, 4(4), pp. 437-455.