A hybrid timing analysis method based on the isolation of software code block

Reference:
Li Haoxuan, De Meulenaere Paul, Vanherpen Ken, Mercelis Siegfried, Hellinckx Peter.- A hybrid timing analysis method based on the isolation of software code block
Internet of Things - ISSN 2542-6605 - 11(2020), 100230
Full text (Publisher's DOI): https://doi.org/10.1016/J.IOT.2020.100230
To cite this reference: https://hdl.handle.net/10067/1709020151162165141
A Hybrid Timing Analysis Method Based on the Isolation of Software Code Block

ARTICLE INFO

Keywords:
hybrid timing analysis
worst-case execution time
WCET
real-time systems
RTOS

ABSTRACT

Estimating the timing behaviour of a system at the early design stage plays a crucial role in reducing the design cost of real-time embedded systems. Various hybrid timing analysis methodologies have been developed aiming to deliver good quality timing analysis results within limited amount of cost. In this paper, we introduce a hybrid approach based on the isolation of software code blocks. This approach provides the possibility for the user to adjust between the quality of the timing analysis results and the cost of the timing analysis according to the user requirements. To improve the results, this approach focuses on three aspects: (i) the consistency between the measured code and the original code under operation; (ii) the test-input data used for measurement; (iii) the state of the hardware prior to the measurement.

1. Introduction

Real-time embedded systems are widely adopted in applications such as automotive, avionics and medical care. Many of these systems are required to meet strict deadlines. Obtaining the timing behaviour of such systems at the early design stage is essential for reducing the design cost and time-to-market. Timing analysis is the process of deriving or estimating the timing properties of a system. Various timing analysis techniques have been developed during the past years, which can be categorized into static, measurement-based and hybrid methods. Hybrid methods combine both static and measurement-based approaches in order to provide quality timing analysis results within a reasonable cost.

Current state of the art hybrid timing analysis approaches need to instrument the original code to extract the timing information. However, the timing behaviour of the program might change due to the instrumentation, which is known as probe effect.

In this paper we introduce a hybrid timing analysis method that is based on a block-isolation technique. This block-isolation technique decouples the execution of a code block from the rest of the program. The size of the blocks can be adjusted at different levels of granularity according to the requirements of the user. The proposed hybrid approach provides the possibility to adjust between the quality (the soundness of the results) and the cost (the efforts and time spending on deriving the results). To improve the results, this approach focuses on three aspects: (i) the consistency between the measured code and the original code under operation; (ii) the test-input data used for measurement; (iii) the state of the hardware prior to the measurement.

The paper is organised as follows: Section 2 gives a brief background of timing analysis techniques. The motivation, problem and contribution is demonstrated in Section 3. The proposed hybrid-timing analysis approach is explained in Section 4. The case study and the experiment setup are elaborated in Section 5. The results and discussion is covered in Section 6 followed by the conclusion and the future work.

2. Background

Timing analysis techniques can be divided into three main categories: static timing analysis, measurement-based timing analysis and hybrid timing analysis. Regarding to the type of analysis results, all the above techniques are divided into two classes: deterministic and probabilistic. The former gives a value of the estimated worst-case execution time (WCET), whilst the latter derives a distribution profile known as probabilistic WCET (Wartel, Kosmidis, Lo, Triquet, Quinones, Abella, Gogonel, Baldovin, Mezzetti, Cucu et al., 2013).

Two requirements that any timing analysis methodology needs to fulfill are safety and precision (Wilhelm, Ebrahim, Ermedahl, Holsti, Thesing, Whalley, Bernat, Ferdinand, Heckmann, Mitra et al., 2008). The former means the estimated time bound must exceed the longest execution time that can occur during the real execution. The latter indicates the tightness between the estimated time bound and the actual WCET value.
2.1. Static timing analysis

Static timing analysis derives the upper execution time bound of a program without the need of executing the program nor conducting a simulation (Wilhelm et al., 2008). During this type of analysis, a timing model of the application is constructed. Prior to this, required information must be gathered by analysing the code and the hardware. The more meticulous the model is, the more precise the estimated WCET will become.

2.2. Measurement-based timing analysis

Measurement-based timing analysis is also referred as dynamic timing analysis or statistical timing analysis. In contrast to static timing analysis, measurement-based timing analysis estimates the execution time by running the program on the target hardware. Considerable amount of tests must be conducted to detect the worst-case scenario.

2.3. Static timing analysis VS measurement-based timing analysis

Static timing analysis is preferred by the industry when it comes to simple applications. Due to the maturity of timing analysis techniques for uninterrupted execution on single core platforms, most of the control modules in critical applications such as avionics are still done on single core (Nowotsch and Paulitsch, 2012). However, the increased performance requirements and the intention of reducing the number of electronic control units (ECU) used in one system have given rise to the multi-core systems (Navet, Monot, Bavoux and Simonot-Lion, 2010; Tabish, Mancuso, Wasly, Alhammad, Phatak, Pellizzoni and Caccamo, 2016). This evolution leads to the increase in the complexity of timing analysis. It is no longer feasible to derive sound and reliable results within a tolerable cost and effort using static timing analysis (Wilhelm and Reineke, 2012).

Compared with static timing analysis, measurement-based timing analysis requires considerably less effort to implement. Since measurement-based timing analysis replaces hardware behaviour analysis by actual measurement, the effort to deal with complicated hardware is rather small. However, in measurement-based timing analysis, only a subset of all possible executions is measured. Therefore, it is never guaranteed that the worst-case scenario leading to the longest execution time can be covered by the measurement (Wilhelm et al., 2008). In the state of practice, a safe margin is usually added to the final result to increase the confidence of the estimated WCET. However, the margin is purely empirical without theoretical backup. Consequently, the results are not as sound as the ones derived by static timing analysis.

2.4. Hybrid timing analysis

Hybrid timing analysis combines both static and measurement-based techniques in estimating the WCET. Unlike measurement-based timing analysis, in which exhaustive execution of end-to-end program paths is computationally infeasible in most cases (Wenzel, Kirner, Rieder and Puschner, 2008), in hybrid timing analysis, measurements are performed on program segments instead of the complete program (Petters and Farber, 1999; Wilhelm et al., 2008; Wenzel et al., 2008; Gustafsson, Betts, Ermedahl and Lisper, 2010). Instrumentation Points (Ipoints) are inserted around these segments to collect the time stamps at these particular program points. The impact of different ways of instrumentation are discussed by Petters (2003). After the instrumented program undergoes intensive executions, a set of traces is collected. Every trace consists of a a sequence of tuples. Every tuple contains an Ipoint identifier and the time stamp at the point of execution (Betts and Donaldson, 2013). Timing information such as the WCET of every segment can be extracted from the traces. The WCET of the global program is eventually constructed in a static way, based on the control flow model of the program.

3. Problems and contribution

3.1. Impact of instrumentation

To date, various hybrid timing analysis methods have been developed and introduced such as the ones proposed by Petters (2000); Bernat, Colin and Petters (2002); Wenzel et al. (2008); Huybrechts, Bock, Li and Hellinckx (2019) and Betts and Donaldson (2013). As explained in the previous section, these techniques rely on instrumentation to extract timing information. However, the timing behaviour of the program can change due to the instrumentation, which is known as probe effect. The influence of instrumentation on the hardware state during the execution is widely known. On the other hand, the influence of instrumentation on the software during compilation has not received enough attention. The example in Figure 1 shows that after compilation, the instrumented code is no longer the same as the original code at assembly level. To avoid the probe effect, Betts, Merriam and Bernat (2010) and Dreyer, Hochberger,
C:
```c
fir2dim_pin_down( &fir2dim_image[0], &fir2dim_array[0], &fir2dim_coefficients[0], &fir2dim_output[0] );
```

Assembly:

<table>
<thead>
<tr>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r5, [pc, #228]</td>
<td>ldr r7, [pc, #636]</td>
</tr>
<tr>
<td>ldr r4, [pc, #232]</td>
<td>ldr w r8, [pc, #664]</td>
</tr>
<tr>
<td>ldr r6, [pc, #232]</td>
<td>mov r3, r7</td>
</tr>
<tr>
<td>mov r3, r5</td>
<td>ldr r2, [pc, #632]</td>
</tr>
<tr>
<td>mov r2, r4</td>
<td>mov r1, r8</td>
</tr>
<tr>
<td>mov r1, r6</td>
<td>ldr r0, [pc, #632]</td>
</tr>
<tr>
<td>ldr r0, [pc, #228]</td>
<td>ldr w r9, [pc, #648]</td>
</tr>
<tr>
<td>ldr r7, [pc, #232]</td>
<td>blx r9</td>
</tr>
<tr>
<td>blx r7</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1:** Original C code and Instrumented C code at assembly level. Compiler: Arm Embedded GCC Compiler; Optimize: -O1.

Lange, Wegener and Weiss (2016) proposed non-intrusive hybrid methods using hardware tracing units such as on-chip debug interfaces to record the execution information of the program. Nevertheless, these methods are only limited to processors that provide tracing facilities.

Our method reduces the impact of instrumentation on the program by dividing and measuring the blocks at assembly level, which guarantees the consistency between the measured code and the original code under operation.

### 3.2. Test-input data

The test-input data is extremely critical for measurement-based and hybrid timing analysis, as it directly decides which execution path will be covered during the measurement. (Wenzel et al., 2008) suggested to improve the coverage rate of measurement by strategically generating input data for testing using model checking. However, this method has two major drawbacks. First, model checking is computationally expensive. Second, coverage rate cannot be used to indicate whether the WCET is covered. For example, when a processor does not contain any floating-point units, the execution iteration of floating-point operation is determined by the values of floating-point inputs. Test-input data reaching 100% code coverage rate does not necessarily mean the worst-case scenario is observed during the measurement. Hence, how to generate the test-input data within a reasonable amount of cost remains a challenge for timing analysis.

In our method, the isolated blocks can be executed independently from the rest of the program. Dedicated input can be generated in a systematic fashion for every block. In general, the complexity of input generation reduces with smaller block size. The reason is that less inputs are usually required by smaller blocks, which reduces the dimension of search space for input generation.

### 3.3. Influence of hardware

The hardware state plays a major role in complex processors with acceleration features such as cache and pipeline. If a context switch occurs, the cache can be filled with irrelevant instructions and data, which can increase the execution time of the following instructions due to the cache miss penalty. Thus, measuring the execution time without considering the hardware state may lead to underestimation in the result. In order to measure the execution time under different hardware states for WCET analysis, additional instructions need to be inserted to adjust the hardware state. Therefore, non-intrusive techniques proposed in Betts et al. (2010) and Dreyer et al. (2016) are not able to deal with this situation. On the other hand, for hybrid methods that based on instrumentation, adding instructions to adjust the hardware states before every block in the complete program can result in drastic overall code size increase. Additionally, it is difficult to investigate a specific code block without influencing the rest of the program.

In our method, because of the block-isolation technique, the processor state can be easily adjusted or created specifically for one component. For example, when cache is present, we can fill the cache with irrelevant instructions and data to force the component to execute under a processor state that can lead to the worst-case scenario.
3.4. Trade-off between the quality of the results and the cost

The purpose of hybrid timing analysis is combining both static and measurement-based techniques to generate quality timing analysis within a reasonable cost. In practice, different applications usually have different requirements regarding the quality of the results and the cost spent on timing analysis. However, most of the current hybrid timing analysis techniques cannot be tuned accordingly to the user requirements. Eventually, either the final result is not reliable enough for the application or unnecessary efforts and resources are spent on the timing analysis.

Using our method, programs can be decomposed into different sized blocks at different levels. The concept of decomposition level is explained in section 4.2. The block size and the number of blocks of a program can be adjusted by changing the decomposition level. This can help the user to find the balance between the quality and the cost of the results according to the requirements.

4. Proposed hybrid timing analysis technique

The proposed hybrid timing analysis method is explained in this section. First, we will explain the detailed information of how to divide the original code and the block-isolation technique adopted in this method. Using this technique, the execution time of the blocks is measured at assembly level which guarantees the consistency between the measured code and the original code by reducing the impact of instrumentation. The processor state can be adjusted and studied for a specific block without significantly increasing the overall block size. Test-input generation is covered in the end of this section.

4.1. Assembly block extraction

![Figure 2: The process to extract assembly blocks.]

It is important to clarify that the block we created in our method is not same as the basic block defined by Lokućiejeewski and Marwedel (2010), which has only one entry at the beginning and one exit at the end of the block. Depending on the content, the blocks in our case can have one or multiple exits anywhere in the block.

To decouple the execution of a block from the rest of the program, our method divides the program into blocks at assembly level to guarantee the consistency between the measured code and the original code. The process is illustrated in Figure 2. Based on the requirements of the user, the original program is divided into small C-code blocks at the required decomposition level. The start and end flags are inserted around the blocks for tracking the corresponding assembly code after compilation. By the end of this process the C-code blocks are converted into corresponding assembly-code blocks. At this stage, a timed automata model expressing the causal relationships between the blocks of the program is constructed automatically.

4.2. Block size and decomposition level

One major advantage of our method is that the blocks can be generated at different decomposition levels according to the user requirement. Figure 3 is an example of a program at decomposition level 1 and decomposition level 2. At decomposition level 1, the program is decomposed by opening the most peripheral brackets. The second most peripheral brackets are opened at decomposition level 2. Generally speaking, with increased decomposition level, more blocks are produced, the average size of a block becomes smaller, less inputs are required for every block, more paths are revealed in the constructed model.
A Hybrid Timing Analysis Method Based on the Isolation of Software Code Block

```c
int fvalue, mid, up, low;
low = 0;
up = 14;
fvalue = -1;     // Block 1

while (low <= up) {
    mid = (low + up) >> 1;
    if (binarysearch_data[mid].key == x) {
        up = low - 1;
        fvalue = binarysearch_data[mid].value;
    } else if (binarysearch_data[mid].key > x) {
        up = mid - 1;
    } else {
        low = mid + 1;
    }
} return(fvalue);  // Block 3
```

```c
int fvalue, mid, up, low;
low = 0;
up = 14;
fvalue = -1;     // Block 1

while (low <= up) {
    mid = (low + up) >> 1;
    if (binarysearch_data[mid].key == x) {
        up = low - 1;
        fvalue = binarysearch_data[mid].value;
    } else if (binarysearch_data[mid].key > x) {
        up = mid - 1;
    } else {
        low = mid + 1;
    }
} return(fvalue);  // Block 3
```

**Figure 3:** Blocks of a program at decomposition level 1 and decomposition level 2.

### 4.3. Block-isolation technique implementation

With the goal of timing analysis, the processor state $S$ can be represented by the values in registers $Vr$, memory $Vm$, stack $Vs$ and the state of the acceleration features (e.g. cache $Sc$, pipeline $Sp)$:

$$S(Vr, Vm, Vs, Sc, Sp, ...)$$

The processor state prior to the execution of block during execution is as $S(Vr_0, Vm_0, Vs_0, Sc_0, Sp_0, ...)$ in which $Vr_0, Vm_0, Vs_0$ are the values in registers, memory space, stack $Sc_0$ and $Sp_0$ are the states of cache and pipeline before the execution of the block. The key of the block-isolation technique is to set the processor state to $S(Vr’, Vm’, Vs’, Sc_{wcest}, Sp_{wcest}, ...)$ before measuring the execution time of the block. $Vr’, Vm’, Vs’$ are the registers, memory space and stack with test inputs. The test inputs are generated specifically for the block. The information regarding test-input data generation is covered further in section 4.4. $Sc_{wcest}$ and $Sp_{wcest}$ are the state of the these acceleration features at which the worst-case scenario has the highest chance to occur. For example, in this paper, we assume $Sc_{wcest}$ is the state at which the caches are filled with irrelevant instructions and data, $Sp_{wcest}$ is the state at which the pipeline is just flushed.

**Figure 4:** Key components of the proposed hybrid timing analysis method.
Figure 4 illustrates the key components of the proposed hybrid timing analysis method. The implementation of the block-isolation technique includes Stack Length Detection, Input Value Update and Processor State Initialisation.

### 4.3.1. Stack length detection

![Stack Length Detection Diagram](image)

Figure 5: Implementation of stack length detection.

Since the values in stack will be stored and modified in a reserved memory space, the stack length needs to be detected to determine the size of the memory space (Figure 5). This process is achieved using assembly instructions to collect the values stored in the stack pointer and the frame pointer. The length of the stack can then be calculated by subtracting the stack pointer value from the frame pointer value. Depending on the target processor, the stack length can also be calculated by analysing the assembly code.

### 4.3.2. Input value update

![Full Execution of Program Diagram](image)

Figure 6: The role of the full execution of the program before input value update.

Before updating the input values, the full program will be executed once (Figure 6). During this execution, the values in registers ($Vr0$) and the values in stack ($Vs0$) are collected and stored to the reserved memory space using assembly instructions.

![Input Value Update Diagram](image)

Figure 7: Implementation of input value update.

To update the input values during the measurement (Figure 7), we divided the inputs into global inputs and local inputs. Global inputs are the global variables of the complete program, the values of these inputs are stored in memory.
To update the value of a global input, we assign the new value directly to the memory space dedicated to the global input. In contrast, local inputs are the local variables used by the code of the block. Since the value of a local variable is passed on using registers or stacks, we have to identify the registers or stack location used to store the value of the local variable. The process is automated in our approach, the details are elaborated in Figure 8: first, our tool analyses the C code and the variable low is identified as critical input. Critical inputs are the ones that can influence the execution time of a program by changing the program flow, loop bound or through instructions or libraries with indeterminate execution time (e.g. floating-point operation on processors without floating-point unit). After analysing the correspondent assembly code, our tool located that $Y+5$ and $Y+6$ are the stack locations used to store the value of the local variable low ($Y$ is the stack pointer). To update the input value of local variable low, we will assign the generated input value to the corresponding locations in the reserved memory space for stack. After processor state initialisation, the stack will be replaced by the values stored in the reserved memory space for stack. The new input value is therefore assigned to the local variable before the execution of the block. In the proposed hybrid method, we use a heuristic method to strategically generate test-input data, the details are explained in section 4.4.

4.3.3. Processor state initialisation

After the input value update, the reserved memory space is assigned with new input values $(Vr', Vm', Vs')$. During processor state initialisation, the registers $(Vr0)$ and stack $(Vs0)$ are replaced by new input values $Vr'$ and $Vs'$ stored in the reserved memory space (Figure 9). At this stage, the state of the acceleration features can be easily adjusted. For example, the cache can be filled with irrelevant instructions and data to create a cache-miss scenario, and the pipeline can be flushed. The intention is to adjust the hardware to create a state that can cause worst-case scenario.

After processor state initialisation, the processor state is:

$$S(Vr', Vm', Vs', Sc_{ucest}, Sp_{ucest}, ...)$$

with new input values $Vr'$, $Vm'$, $Vs'$ and hardware state $Sc_{ucest}$, $Sp_{ucest}$ prior to execution of the assembly block.

4.4. Test-input data generation

Test-input data plays a significant role in deriving a safe upper bound in timing analysis. However, it is still a challenging task to generate the data needed for testing. In most of the state of the art hybrid timing analysis methodologies,
the input is either provided, or generated for the complete program. In our approach, we propose to thoroughly test every block with inputs generated specifically for the block. This should be particularly beneficial when using heuristic algorithm such as generic algorithm or particle swarm optimization. By reducing the number of inputs, the exploration space of heuristic algorithm is decreased, which ultimately reduces the complexity of input generation.

To reduce the number of inputs needed to be generated, we only consider critical inputs. Critical inputs are the ones that can influence the execution time of a program by changing the program flow, loop bound or through instructions or libraries with indeterminate execution time (e.g. floating-point operation on processors without floating-point unit).

The first step to identify critical inputs is to find the input variables that are independent from other variables. For example in operation: \( x = a + b \), the value of \( x \) is dependent on variables \( a \) and \( b \). Therefore, we consider variable \( x \) as a non-critical input. If \( a \) and \( b \) are not dependent on any other variables, \( a \) and \( b \) are considered as candidates to be critical inputs. The critical input variables are a subset of those candidates including the ones appeared in conditional instructions, loop bounds, function calls and the floating-point variables (for processors without floating-point unit).

It is important to point out that hybrid-timing analysis is extremely vulnerable to invalid input values (the input values that can never occur during execution of the program). This is because invalid input may potentially lead to the execution of an infeasible path which brings pessimism to the final result. As illustrated in Figure 10, domain specific knowledge or program analysis is applied in this process to generate annotations to narrow down the value range of the inputs to remove invalid inputs and to reduce the search space.

In heuristic input generation, the result of every measurement is represented by tuple \( (D_i, T_i) \), where \( i \) is the index of the measurement, \( (D_i) \) is the input data used for the block, \( (T_i) \) is the measured execution time of the block. The tuples generated by the previous measurement are used by the heuristic algorithm to optimize the generation of the upcoming input data.

For the scope of this paper, we choose particle swarm optimization (PSO) to validate our method because PSO is designed to search for global maxima which is compliant with the goal of WCET analysis. Another reason is the efficient searching speed (Rini, Shamsuddin and Yuhaniz, 2011). However, our input generation is not limited to PSO, algorithms can be implemented easily using COBRA Input Generator provided by the COBRA-framework. Other heuristic methods will be investigated in the future research.

### 4.5. Recursion

Recursion has been a challenge in hybrid timing analysis due to the instrumentation used to collect traces. The
reason is that the impact of the inserted instructions will accumulate through the recursion, which leads to drastic increases in the measured execution time.

Our technique solves the problem by redirecting the execution flow to a duplicated recursion function, in which all the additional instructions are removed. As it is shown in Figure 11, the assembly code of the recursion function is duplicated next to the assembly block. The function call distance and jump distance of the block is adjusted to the duplicated recursion function instead of the original recursion function. By doing this, once the call to the recursion function is triggered, the execution will continue at the start of the duplicated recursion function. When the recursion operation is completed, the program flow will continue in the assembly block. This implementation is achievable because the blocks can execute independently from the rest of the program.

4.6. Calculating the Overall WCET

After the execution times of all the components are collected, the static approach is applied to derive the estimated timing information. At this stage, a timed automata ([Alur and Dill, 1994; Larsen, Pettersson and Yi, 1997]) model expressing the causal relationships between the components of the system is constructed. The execution time can be calculated using the Implicit Path Enumeration Technique (IPET) (Engblom, Ermedahl, Sjödin, Gustafsson and Hansson, 2003).

![Timed automata of an example program](a) Timed automata of an example program

![IPET calculation of the example program](b) IPET calculation of the example program

Figure 12: WCET calculation example.

Figure 12 is one example of how to use the IPET technique to calculate the WCET. Fig. 12(a) is the timed automata of the example program. The A\(3\) indicates the execution time of automaton A is 3 clock cycles. Fig. 12(b) translates the timed automata for the IPET calculation. XA indicates the number of times the timed automaton A has been executed. XAB represents the number of times the transition from A to B is taken. The search of the WCET is expressed as an integer linear programming program in which the goal is to maximize the execution time of the program under constraints. By default both the start and exit constraints are given the value 1 because a program will start and exit once for every execution. The structural constraints can be interpreted as such: the times one automaton is executed is equal to the sum of the executions from all the preceding automata and the sum of all the executions to the descending automata. The loop-bound constraints is derived from the maxiter. Eventually the WCET can be calculated using the cost function in which every automaton is given a time coefficient.
5. Case study

5.1. Case study

A case-study approach was adopted to evaluate the proposed hybrid timing analysis method. TACLeBench (Falk, Altmeyer, Hellinckx, Lisper, Puffitsch, Rochange, Schoeberl, Sørensen, Wägemann and Wegener, 2016) was employed in the design of the case study. To obtain in-depth knowledge on the impact of the decomposition of a program, we conducted the case study at 3 decomposition levels. Because the inputs are generated for every block in our method, it is not possible to compare results using the same inputs with commercial tools such as RapiTime (Systems, 2008). The execution time of the complete benchmark is measured with input data generated by COBRA Input Generator. The input data that produced the longest observed execution time is then used by RapiTime to compare with the execution time measured using the complete benchmark.

Due to the limited memory space of the processor, we tested the proposed technique on 10 benchmarks. Every block was executed 500 times respectively for random input generation and particle swarm optimization (10 particles; 50 iterations). A new set of inputs is used in every execution. The complete benchmark was measured 500 times as well in both cases.

To fairly compare the results of our methodology with RapiTime, we turned off the compiler optimization so that the measured code is identical at the assembly level.

The case study is designed aiming to validate our method by answering the following questions:

1. Is it possible to decompose a program and perform measurements at assembly-block level?
2. Is it beneficial to generate test-input data for every block using a heuristic method?
3. What is the possible impact of instrumentation when adjusting the hardware state and how to reduce it?
4. What is the influence of the decomposition level on the quality of timing analysis results?

5.2. Experiment setup

![Experiment setup diagram](image)

The experiment setup mainly consists of an FPGA and a processor as illustrated in Figure 13. To answer the above questions, we adopted two processors. Generally speaking, the complete case study can be considered as two independent parts.

The first part aims at validating the experiment results with generated inputs at different decomposition levels. We selected an Atmel AT90CAN128 single-core processor which has a very simple architecture without any complicated acceleration features. This eliminates the influences of hardware, which makes it easier to compare the results measured using different input generation methods. In this study we choose random and PSO because one of the goals of the case study is to compare random input generation with some heuristic method. It is worthy of mentioning that PSO is merely an arbitrary choice from the set of available heuristic methods. To investigate which heuristic method has the best performance is out of the scope of this paper, further research is required to cover this topic.

The second part is focusing on the influence of the hardware state at different decomposition levels. In this part of case study, we adopted a 32-bit Cortex-M7 single-core processor in order to reduce the impact of inputs from the results. Since the Cortex-M7 processor contains floating-point units (FPU), the execution time is no longer influenced by the floating-point operation (indeterminate execution time on processors without FPU).

To measure the execution time, we used a Xilinx Nexus 3 FPGA. This setup decouples the timing measurement from the target processor, which minimizes the effort to migrate to other processors. As can be seen in Figure 13, the
A Hybrid Timing Analysis Method Based on the Isolation of Software Code Block

Table 1
Measured WCET with AT90CAN128. cc: clock cycles; DL: decomposition level; NoB: number of blocks; CP: execution time measured using complete benchmark; RT: result of RapiTime; RAN: random input generation; PSO: particle swarm optimization input generation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DL1 (cc)</th>
<th>NoB</th>
<th>DL2 (cc)</th>
<th>NoB</th>
<th>DL3 (cc)</th>
<th>NoB</th>
<th>CP (cc)</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>iir</td>
<td>4969</td>
<td>5170</td>
<td>5188</td>
<td>5248</td>
<td>-</td>
<td>-</td>
<td>4955</td>
<td>5086</td>
</tr>
<tr>
<td>fac</td>
<td>1328</td>
<td>1328</td>
<td>2183</td>
<td>2183</td>
<td>2</td>
<td>-</td>
<td>1328</td>
<td>1328</td>
</tr>
<tr>
<td>complex_updates</td>
<td>19672</td>
<td>19656</td>
<td>19940</td>
<td>19940</td>
<td>4</td>
<td>-</td>
<td>19632</td>
<td>19655</td>
</tr>
<tr>
<td>recursion</td>
<td>10312</td>
<td>10312</td>
<td>10306</td>
<td>10306</td>
<td>5</td>
<td>-</td>
<td>10306</td>
<td>10306</td>
</tr>
<tr>
<td>countnegative</td>
<td>1358</td>
<td>1355</td>
<td>1360</td>
<td>1360</td>
<td>4</td>
<td>1360</td>
<td>1306</td>
<td>1352</td>
</tr>
<tr>
<td>fir2dim</td>
<td>63395</td>
<td>63555</td>
<td>63647</td>
<td>63751</td>
<td>7</td>
<td>65423</td>
<td>65607</td>
<td>63292</td>
</tr>
<tr>
<td>prime</td>
<td>5210</td>
<td>5210</td>
<td>5256</td>
<td>5256</td>
<td>5</td>
<td>5224</td>
<td>5224</td>
<td>5224</td>
</tr>
<tr>
<td>matrix1</td>
<td>54263</td>
<td>54263</td>
<td>54263</td>
<td>54263</td>
<td>4</td>
<td>54263</td>
<td>54263</td>
<td>54263</td>
</tr>
<tr>
<td>binarysearch</td>
<td>385</td>
<td>385</td>
<td>386</td>
<td>386</td>
<td>6</td>
<td>386</td>
<td>385</td>
<td>385</td>
</tr>
<tr>
<td>insertsort</td>
<td>6662</td>
<td>6662</td>
<td>11185</td>
<td>11185</td>
<td>12</td>
<td>11212</td>
<td>11212</td>
<td>6662</td>
</tr>
</tbody>
</table>

processor is driven by an external clock signal generated by the FPGA. The measurement process starts by sending the Start signal from the FPGA to the processor. Two GPIO pins of the processor are used to set the Tic and Toc flags. Before the assembly-code block starts to execute, the Tic pin is toggled to enable the Tic flag. Once the Tic flag is triggered, a timer starts to count the number of clock cycles have passed. After the assembly-code block is executed, the Toc pin is set to enable the Toc flag. The timer counter is then stopped after the Toc is enabled. Eventually, the value in the timer (Execution Time) is collected through a serial port for further analysis.

6. Results and discussion

In the beginning of this section, the results of the case study are presented and discussed in the same order as the questions we raised in section 5. Since the impact of decomposition level is closely involved in all the tests we conducted, the influence of decomposition level on the quality of timing analysis results and the cost is discussed through all the case studies. A general discussion is presented at the end of this section. Because processor AT90CAN128 has a very simple architecture, the overhead of timing measurement (time spent on toggling the pins) is constant. It can be safely removed from the results. However, for processor Cortex-M7 the overhead is not constant due to the acceleration features. Only the shortest overhead was removed. For some blocks, residues of overhead may be left in the results.

6.1. Results validation at different decomposition levels

Table 1 shows the estimated WCETs on AT90CAN128 using our hybrid method. Generally speaking, the estimated WCETs at DL1 and CP are very close. fac, matrix1 and insertsort showed exactly the same result between DL1 and CP. This is because at DL1, only the most peripheral brackets are opened to generate blocks. The impact of the block size and input generation on the measured WCET is still subtle at this decomposition level.

Benchmark matrix1 showed exactly the same estimated WCET at all levels as well as CP and RT. Further investigation revealed the reason is that the execution time of this benchmark is independent from the input. Since the architecture of this processor does not have influence on the execution time, the result stayed the same through all the decomposition levels. This proves that our method is possible to reconstruct the timing behaviour of a program by the timing information measured from assembly blocks.

The results at DL2 and DL3 are usually higher compared with CP. Most of the increase such as in iir, complex updates, countnegative and fir2dim is around 5%. However, in fac and insertsort the increase reached around 50%. This drastic increase can also be observed in RapiTime. The reason is because, similar to RapiTime, we take the assumption that all blocks always execute with the longest measured execution time. The intention of this assumption is meant to produce a safe estimated WCET, however it has also become one major source of pessimism in the final result. The
A Hybrid Timing Analysis Method Based on the Isolation of Software Code Block

Table 2
Measured WCET of blocks in clock cycles at decomposition level 3 with benchmark *fir2dim* using longest-observed-WCET input on processor AT90CAN128. B: block with the starting line number; CP: result of the corresponding blocks in the complete benchmark using longest-observed-WCET input generated for the complete benchmark;

<table>
<thead>
<tr>
<th>Block</th>
<th>B164</th>
<th>B172</th>
<th>B175</th>
<th>B178</th>
<th>B179</th>
<th>B187</th>
<th>B190</th>
<th>B195</th>
<th>B198</th>
<th>B202</th>
<th>B204</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL3</td>
<td>14</td>
<td>2</td>
<td>90</td>
<td>114</td>
<td>49</td>
<td>996</td>
<td>1002</td>
<td>1042</td>
<td>1</td>
<td>431</td>
<td>15102</td>
</tr>
<tr>
<td>CP</td>
<td>14</td>
<td>2</td>
<td>90</td>
<td>114</td>
<td>49</td>
<td>984</td>
<td>982</td>
<td>982</td>
<td>1</td>
<td>318</td>
<td>15102</td>
</tr>
</tbody>
</table>

estimated WCET of the program calculated using the blocks is extremely vulnerable to this type of pessimism when iteration is present.

Another source of pessimism is the concealed infeasible path. Slight decrease was observed in *recursion* and *prime* at DL2 and DL3 respectively. As can be seen in Figure 14, the branch execution sequence of the *recursion* can be clearly reflected at decomposition level 2, however this information is not revealed at decomposition level 1. The execution path shown at decomposition level 1 is an infeasible path as the value of *i* cannot be both 0 and 1 at the same time. Therefore, the quality of the results is improved at a higher decomposition level because the static knowledge of the program is more refined.

In benchmark fac and recursion, the estimated WCET increased in RapiTime compared with our approach. The reason is because both of these benchmarks involve recursion operation. The impact of instrumentation accumulated through the operation which resulted in the drastic increase. Our method solved the problem of recursion by removing the impact of instrumentation during the measurement (Section 4.5).

6.2. The results of using inputs generated specifically for every block

This part focuses on the impact of generating test-input data specifically for every block. First, we compared the results of our method using inputs generated for every block and the results of RapiTime (RT) using the longest-observed-WCET input generated for the complete benchmark (CP) (Table 1). The results stayed the same for benchmarks whose execution time is not dependent on the inputs (prime, matrix1, binarysearch, insertsort). It can also be observed that the results were very close for benchmark *iir* and *countnegative*. In contrast, the results of our method is higher at DL3 compared with RapiTime in complex updates and *fir2dim*. Careful investigation revealed the reason is because the maximum number of inputs required by the block in *iir* and *countnegative* is the same as the complete program. In complex updates and *fir2dim*, the maximum number of inputs is decreased by dividing the program into smaller blocks, which reduces the search space of heuristic algorithm.
This part focuses on the impact of generating test-input data specifically for every block. Table 2 is the results of benchmark fir2dim at decomposition level 3 and the corresponding execution time measured from the complete benchmark using longest-observed-WCET input generated for the complete program. The influence of reducing the number of inputs (search dimension of heuristic algorithm) can be clearly reflected in this table. The difference in the execution time at $B_{187}$, $B_{190}$, $B_{195}$ and $B_{202}$ is because the number of inputs decreased from 4 to 1 which reduced the search dimension of PSO. This reaffirms our previous conclusion that it is possible to improve test-input data by reducing the number of inputs using smaller blocks at high decomposition level.

![Figure 15: Measured WCET of fir2dm on AT90CAN128.](image1)

As can be seen in Figure 15a and Figure 15b, the speed of convergence towards the absolute WCET increases with the decomposition level. The reason is that the input search space is decreased by reducing the number of average inputs required by every block. This is clearly reflected by the result of $CP$ and $DL3$ in Figure 15b: much less iterations are needed for $DL3$ to converge towards the actual WCET compared with $CP$.

![Figure 16: Results of every single execution of blockR178 in fir2dim on AT90CAN128.](image2)

Figure 16a and Figure 16b are the results of single executions of blockR179 in fir2dim using input generated by random and PSO respectively. Unlike the results collected with randomly generated inputs, the results of PSO show visible convergence towards a certain execution time. It is important to point out that random input generation reached
Table 3
Measured WCET of fir2dim blocks on Cortex-M7 with cache disabled at decomposition level 1. The values are in clock cycles. NoN: number of nop instructions; Rxxx: index of the block; Overhead: execution time of the instructions required to extract the execution time of a block.

<table>
<thead>
<tr>
<th>NoN</th>
<th>R164</th>
<th>R169</th>
<th>R172</th>
<th>R175</th>
<th>R202</th>
<th>R204</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>81</td>
<td>5379</td>
<td>57</td>
<td>14039</td>
<td>161</td>
<td>5379</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>71</td>
<td>5511</td>
<td>57</td>
<td>13749</td>
<td>159</td>
<td>5511</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>5503</td>
<td>57</td>
<td>14043</td>
<td>161</td>
<td>5503</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>84</td>
<td>5399</td>
<td>54</td>
<td>14158</td>
<td>178</td>
<td>5399</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>88</td>
<td>5410</td>
<td>54</td>
<td>14386</td>
<td>178</td>
<td>5410</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>80</td>
<td>5180</td>
<td>68</td>
<td>14254</td>
<td>166</td>
<td>5180</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>80</td>
<td>5174</td>
<td>70</td>
<td>13877</td>
<td>168</td>
<td>5173</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>79</td>
<td>5543</td>
<td>63</td>
<td>15749</td>
<td>164</td>
<td>5543</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>83</td>
<td>5557</td>
<td>63</td>
<td>15421</td>
<td>163</td>
<td>5557</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>77</td>
<td>5291</td>
<td>64</td>
<td>13645</td>
<td>163</td>
<td>5291</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>78</td>
<td>5294</td>
<td>64</td>
<td>14429</td>
<td>164</td>
<td>5293</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>94</td>
<td>5292</td>
<td>60</td>
<td>14099</td>
<td>168</td>
<td>5292</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>5306</td>
<td>62</td>
<td>13856</td>
<td>168</td>
<td>5306</td>
<td>36</td>
</tr>
<tr>
<td>13</td>
<td>78</td>
<td>5376</td>
<td>76</td>
<td>15026</td>
<td>162</td>
<td>5376</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>78</td>
<td>5359</td>
<td>76</td>
<td>14728</td>
<td>164</td>
<td>5359</td>
<td>38</td>
</tr>
<tr>
<td>15</td>
<td>77</td>
<td>5370</td>
<td>76</td>
<td>13327</td>
<td>164</td>
<td>5370</td>
<td>38</td>
</tr>
</tbody>
</table>

its measured WCET at around the $200^{th}$ execution, and PSO input generation reached measured WCET 100 executions later at around the $300^{th}$ execution. However, the measured WCET is higher in PSO compared with random. The reason is that the initial input data of PSO are randomly generated. It usually takes some number of executions for PSO to create an input set leading to execution time higher than random. It is also observed in Figure 15 by comparing the results at the same decomposition levels. The measured WCET of PSO quickly passed the measured WCET of random after the initially 100 executions and continued to show stronger convergence towards the absolute WCET across all 4 levels.

Table 1 reveals that using the same number of measurements, the WCET measured using PSO is higher compared with the WCET measured using random. This observation indicates that based on the previous input and execution time, heuristic methods such as PSO can be used to improve the quality of the input.

6.3. Impact of instrumentation on hardware regarding to the timing analysis results

In this part, we conducted our experiment on the ARM Cortex-M7 processor. As mentioned in section 5, the floating point unit of this processor removes the impact of floating point operation on the execution time. Additionally, because of the simplicity of the benchmark, we were able to find the input data produces the measured WCET of the benchmark. These inputs are used in this study to remove the influence of input data. Therefore, the changes in the execution time is only due to the changes in the hardware state, which reduces the efforts to investigate the impact of hardware.

One major influence of instrumentation which is usually ignored is the alignment of the program code in memory. In this paper, we refer the term code alignment as the location of the starting line of a code segment in the memory. Typically for embedded systems, once the code is compiled, the alignment is fixed. However, the instrumentation inserted into the original code will produce differences in the alignments between the measured code and the original code under operation. This is not necessarily a problem for simple processors, however, for processors with complicated features it may raise concerns for the final results. Therefore, in this case study, we investigated influence of the code alignment on the execution time of a code block.

Table 3 is the measured WCET of every block of benchmark fir2dim at decomposition level 1. The results were measured on an ARM Cortex-M7 processor with disabled cache. Before the execution, nop instructions were inserted before the block to change code alignment. It can be noted from the table that changes on the code alignment can result in changes on the execution time. It is worth mentioning that during the case study, we noticed the execution...
time starts a new iteration every 16 nop instructions, which means adding 0 nop instruction before the block had the same effect as adding 16 nop instructions. Further investigation indicates this might be a result of the combination of the pipelining feature and the fetch queue of the processor.

### 6.4. Influence of decomposition level on the timing analysis results

![Bar graph of measured WCET of fir2dim on Cortex-M7](image)

**Figure 17:** Measured WCET of fir2dim on Cortex-M7 at three decomposition levels. DL: decomposition level; cc: clock cycles; CP: execution time measured using complete benchmark;

Figure 17 is the execution time of benchmark fir2dim at three different decomposition levels and the complete benchmark. Generally speaking, the estimated WCET increases with decomposition level. As mentioned in the previous subsection, the impact of input data has no influence in this case study. The increase is due to the pessimism added to guarantee the safety of the results: first, we use the longest WCET of the above mentioned 16 different alignments; second, for the safety reason, we only removed the shortest overhead. Therefore, the more blocks the program is divided into, the more pessimism is left in the results.

The drastic increase at DL3 when cache is enabled, is because we always assume the worst-scenario hardware states for all the blocks. This leads to overly pessimistic results when cache is present.

### 6.5. Discussion

The case studies have validated our method at different decomposition levels. By changing the decomposition level, we can change the number of blocks, block size, number of inputs required by every block and the refinement of the static model. Generally speaking, the estimated overall WCET of the system increases with the decomposition level. The cost of timing analysis as well increases with the decomposition level because more blocks needs to be measured at a higher decomposition level and more pessimism is introduced into the final results. On the other hand, with the increased decomposition level, there is a higher chance the estimated WCET will exceed the absolute WCET, which makes the results more reliable.

Using the proposed method, test-input data can be generated for every block individually using a heuristic method. It is especially beneficial when the number of inputs required by the block is reduced. Therefore, it might be advantageous to strategically divide the program such that the average/maximum number of inputs required by the blocks can be decreased.

For modern processors, instrumentation can produce major impact on the analysis results. After instrumentation, the alignment of the code may be changed completely, the measured WCET of a block may no longer be the same as the WCET compared to the code with original alignment. This can raise safety concerns in the final results. In our method, in order to assure the safety of the results, we always assume the worst-case scenario. We use the longest WCET of a block measured with different alignments to calculate the final results. The same applies to the pipeline and the cache, the former is flushed prior to the execution of the block, the latter is always filled with irrelevant instructions and data. The drawback of this assumption is that the introduced pessimism will reduce the precision of the results.
However, when we can fully investigate and control the hardware states, the safety margin that is normally added to the final result to increase the confidence of the estimated WCET may be reduced.

Another source of pessimism is the overhead created by the instructions used to extract the timing information. For processors with a simple architecture, it is safe to remove the overhead by simply subtract it from the results. However, when it comes to advanced processors, the overhead can vary based on the alignment of the code or the execution history. Some method removes the overhead by subtracting the time measured by executing an empty block. Nevertheless, this does not consider the impact of alignment and execution history, which may remove more overhead from the results than needed. For the sake of safety, we only remove the shortest overhead time measured at different scenarios. Unfortunately, part of the overhead may still remain in some blocks. Therefore, to reduce this type of pessimism, we propose to combine small blocks (blocks with deterministic execution time) with the adjacent blocks to reduce the number of blocks.

7. Conclusion

To date, various hybrid timing analysis approaches have been developed. However, those approaches either suffer from probe effects such as changes of the timing behaviour of the measured program due to the inserted instructions, or require hardware support to generate execution traces for further analysis. To overcome these problems, we propose a new approach to reduce the probe effects by dividing and measuring the target program at assembly level to guarantee the measured code stays the same compared with the original code under operation. This approach can be applied to any processor.

We investigated the relationship between the quality of timing analysis results and the cost by conducting measurements at different decomposition levels. By increasing the decomposition level, the cost to derive the analysis results increases, the results becomes more sound/reliable. However, due to the assumption that every block always executes with the measured WCET and the residue of overhead, the results are more pessimistic when the decomposition level increases. We realized that our method can be overly conservative compared with alternative approaches, however, we intend to put the safety of the results at the first priority. Our method may be more suitable to measure the WCET of a specific block when interrupts are enabled. For example, if a context switch occurs before a block, the state of the hardware is mostly unknown. It is then necessary to consider the worst-case scenario for the safety of the program.

Compared with other available methods, we provide the possibility for the user to adjust the decomposition level based on the requirements. The hardware states can be adjusted specifically for a block without drastically increasing the overall code size. Eventually, after achieving certain level of investigation and control of the hardware states in the measurements, the safety margin that is normally added to the final result might be reduced to produce a tighter upper bound.

8. Future Work

One main focus of the future research addresses the reduction of the pessimism in the results for a tighter upper bound. Based on the results of this study, one possible approach is to reduce the number of blocks by combining simple consecutive blocks which do not have any influence on the flow of the program. By doing this, the overhead introduced by the measurement can be reduced.

As this study shows that generating inputs for every block individually is more beneficial when the number of inputs is decreased, for the future work, we propose to divide the program strategically to reduce the number of inputs required by every block to reduce the search space of input generation.

Acknowledgement

This research was supported by Flanders Make, the strategic research centre for the manufacturing industry in Belgium, CSE-Codesign project, and by imec, the world-leading R&D and innovation hub in nanoelectronics and digital technologies.

References

A Hybrid Timing Analysis Method Based on the Isolation of Software Code Block


