COBRA-HPA : a block generating tool to perform hybrid program analysis

COBRA-HPA: a Block Generating Tool to Perform Hybrid Program Analysis

Thomas Huybrechts

University of Antwerp - imec, IDLab, Faculty of Applied Engineering, Belgium
E-mail: thomas.huybrechts@uantwerpen.be

Yorick De Bock

University of Antwerp - imec, IDLab, Faculty of Applied Engineering, Belgium
E-mail: yorick.debock@uantwerpen.be

Haoxuan Li

University of Antwerp - imec, IDLab, Faculty of Applied Engineering, Belgium
E-mail: haoxuan.li@uantwerpen.be

Peter Hellinckx

University of Antwerp - imec, IDLab, Faculty of Applied Engineering, Belgium
E-mail: peter.hellinckx@uantwerpen.be

Abstract: The Worst-Case Execution Time of a task is an important value in real-time systems. This metric is used by the scheduler in order to schedule all tasks before their deadlines. However, the code and hardware architecture have a significant impact on the execution time and thus the WCET. Therefore, different analysis methodologies exist to determine the WCET, each with their own advantages and/or disadvantages. In this paper, a hybrid approach is proposed which combines the strengths of two common analysis techniques. The two-layer hybrid model splits the code of tasks into so-called *basic blocks*. The WCET can be determined by performing execution time measurements on each block and statically combining those results. The COBRA-HPA framework presented in this paper is developed to facilitate the creation of hybrid block models and automate the measurements/analysis process. Additionally, an elaborated discussion on the implementation and performance of the framework is given. In conclusion, the results of the COBRA-HPA framework shows a significant reduction in analysis effort while keeping sound WCET predictions for the hybrid method compared to the static and measurement-based approach.

Keywords: Worst-Case Execution Time; WCET; Hybrid Analysis Methodology; COde Behaviour fRAmework; COBRA; Basic Block Generator.

Reference to this paper should be made as follows: Huybrechts T., De Bock Y., Li H. and Hellinckx P. (xxxx) 'COBRA-HPA: a Block Generating Tool to Perform Hybrid Program Analysis', *International Journal of Grid and Utility Computing*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Thomas Huybrechts is a first-year PhD Student of IDLab, imec at the Faculty of Applied Engineering at the University of Antwerp. He received a M.Sc. in Electronics and ICT Engineering Technology with greatest distinction. His research interests are distributed embedded systems, worst-case resource consumption and IoT.

Yorick De Bock is in his final year of his PhD at IDLab, imec and the Faculty of Applied Engineering at the University of Antwerp. His research interests are real-time systems, scheduling theory and real-time virtualisation.

Haoxuan Li obtained his M.Sc. in Biomedical Engineering from Delft University of Technology. Later, he started Advanced Master Studies in Embedded Systems Design in ALaRI. In 2015 he started a PhD at CoSys-Lab and IDLab, imec at the University of Antwerp. His research focuses on timing analysis on real-time embedded systems.

Peter Hellinckx obtained his Master in Computer Science and his Ph.D. in Science at the University of Antwerp. In 2009, he joined TERA-Labs at Karel de Grote University College where he became senior researcher and responsible for the distributed computing research group. In 2013, he became assistant professor in the faculty of applied engineering at UAntwerp. In 2015, he became head of the Electronics-ICT department and joined the IDLab research team as part of imec in 2016. He co-founded the spin-offs Hysopt and Hi10. His research focuses on distributed and real-time embedded software, cloud computing, IoT, cyber physical systems and agent based simulation.

1 Introduction

Embedded systems have grown more prominent in our environment in the last decade. For instance, it is expected that the number of connected Internet of Things (IoT) devices alone will triple in less than four years and reach the limit of 20 billion units in 2020 [1]. One can find these systems in mobile phones, cars, avionics, etc. With the advent of new technology, such as cyber-physical systems (CPS) and IoT, new constraints to size, cost price, energy consumption and real-time behaviour (i.e. execution time) are imposed on these systems to create cheap, reliable and safe systems on a massive scale. The code running on these devices has an impact on these constraints, therefore system designers need information about the effect of running application code w.r.t. the system constraints. On a single-core processor, these principles are well understood. However, growing demand for embedded systems with more resources has introduced new hardware techniques, e.g. multi-core processors, pipelining, etc. Thus, more sophisticated analysis techniques are needed to gauge the fulfilment of constraints on these complex hardware.

In order to improve the execution time of a program, a variety of optimisation techniques are used to improve the throughput of the processor, such as pipelining, caches, branch prediction, multi-core architectures on hardware level; and pre-emption, parallelisation on software level. These techniques are often used in multi-purpose computer systems, e.g. desktops, laptops and servers. It is not a life threatening issue for these systems that a calculation takes more time to complete in certain situations. In these systems, the average execution time of the code is optimized. However, the Worst-Case Execution Time (WCET) is a very important factor for time-critical systems. For instance, the Electronic Control Unit (ECU) that controls the airbag system of a car should respond to a collision within a strict time frame. Therefore, it is important that these systems are deterministic. Optimisations are necessary in common embedded systems, but most compromise the deterministic behaviour of the software running on those processors, e.g. memory access by caching. In the state of the art, these influences are often neglected or the hardware is overdimensioned to minimize the impact, due to the complexity of the predictability analysis. As a result, powerful and expensive processors are used in systems in which they are not necessary. The designer wants to make sure that the real-time constraints are met. Therefore, he/she will overestimate the WCET by underestimating the benefits of the optimisation techniques mentioned above.

It is important in real-time embedded systems that the tasks are handled within the specified time frame, so that the deadlines of the tasks can be achieved. When using an operating system or a hypervisor to run virtual machines, the scheduler is responsible for analysing a set of tasks to determine if it is schedulable. In the 70's Liu and Layland considered the schedulability of a set of tasks with a lot of assumptions [2]. However, these constraints do not always hold in the design of real-time embedded systems because tasks are rarely independent and often reactive instead of periodic. Later work focuses on extending the work of Liu and Layland to relax the outlined constraints. More recently, Bini et al. improved the bound given by Liu and Layland for the monotonic algorithm [3].

In order to determine the schedulability of a set of tasks, a timing analysis is performed to calculate the WCET of each task. In practice, margins are added to the calculated upper and lower bounds of the timing distribution because of the complexity of the analysis as shown in figure 1. For instance, when taking the influences of the cache into account, a cache miss will always be assumed when calculating the upper bound. This pessimistic attitude is not realistic and will eventually lead to over-proportioned systems [4].

Current WCET analysis methodologies, such as the static and measurement-based approach, have their limitations. A big trade-off needs to be made between the accuracy and computational complexity of the analysis. We believe the solution for this shortcoming in the state of the art, lies in a balance with a hybrid approach. In order to determine the WCET with this hybrid methodology, we are creating the COBRA-HPA framework that allows us to perform and automate the hybrid timing analysis on a wide-range of embedded platforms based on higher level code analysis.

In this paper, we will first discuss the different existing WCET analysis techniques, followed by the hybrid methodology. Second, we present our COBRA framework on which we are able to analyse the behaviour of code on different embedded platforms. Final, we discuss our results and performances of the framework.

2 WCET analysis methodologies

In the current state of the art, there are three main strategies to determine the WCET. Each of these has their own advantages and disadvantages. A first methodology is a *time measurement-based* approach of the WCET. In this method, a limited number of different input sets are given to the system. The measurement results will be distributed, for example, according to figure 1. This distribution leads to three important boundaries: Best-Case (BCET), Average-Case (ACET) and Worst-Case Execution Time (WCET). This solution is computationally acceptable as the amount of measurements is decided by the researcher. The accuracy on the other hand will drop dramatically when the amount of measurements decreases as not all system states are reached in the measurements [5] [6]. Therefore, safety margins need to be taken into account when deducing a WCET from the measured execution times.

A second methodology describes the *static analysis* of the code and architecture to obtain an accurate WCET [5] [6] [7] [8]. Static code analysis can determine

the WCET without executing the code itself. However, detailed models of the application and the used hardware is required to obtain sound results with a low upper bound. In addition, the hardware models are not always publicly available by the manufacturer. For instance, the complexity of the static code analysis increases tremendously for multi-core systems in comparison with single-core systems. The different cores can fetch data simultaneously from the memory. A standard L2-cache, as illustrated in figure 2, shares the memory between the multiple cores of the processor. Each core has access to the entire cache memory. In order to calculate the WCET, each possible state of the cache must be known at each point of all the possible program traces. Since multiple programs with different context can be executed concurrently, all the influences (e.g. pre-emption, shared resources, influences of other tasks on other cores) have to be taken into account in order to obtain a sound approximation of the bounds of the execution time. A static cache analysis generates an immense amount of data and becomes too complex which makes this method infeasible to use for bigger tasks.

In [5], Lv et al. mention the effects of shared caches in multi-core systems on the WCET analysis. The analysis of parallel tasks using shared resources and interfering with on another is very difficult and generates an enormous number of possible resource states that must be taken into account. Furthermore, the interference deteriorates the overall WCET of any task on a multi-core processor. In order to make an accurate and feasible prediction for these systems, it is stated that changes need to be made to hardware and software. As a result, the systems will become more timing predictable.

The last methodology called *timing compositionality* proposes to decompose the processor architecture in minor hardware and software systems, whose timing contributions can be calculated independently from other systems. The upper bound of the execution time is then calculated from all the timing contributions of each subsystem. In order to achieve compositionality in multi-core systems, Hahn et al. presents two different approaches [9].

A first approach is to find a compositional analysis for an existing architecture. The development of a sound compositional timing analysis becomes very complex as the available resources (e.g. multi-level caches) are shared among the different cores as shown in figure 2. The analysis cannot be further decomposed through the different dependencies which influence the timing distribution. Up to now, there is still no compositional decomposition found for any architecture and it will never be possible to perform a decomposition for every currently existing architecture [9]. Nevertheless, modifications can be applied to existing hardware to support decomposition. In [10], Chisholm et al. propose to partition the shared cache such that each core has access to an assigned part of the cache. This allows the shared cache to be decomposable, making it possible to establish a timing distribution of the cache. However,

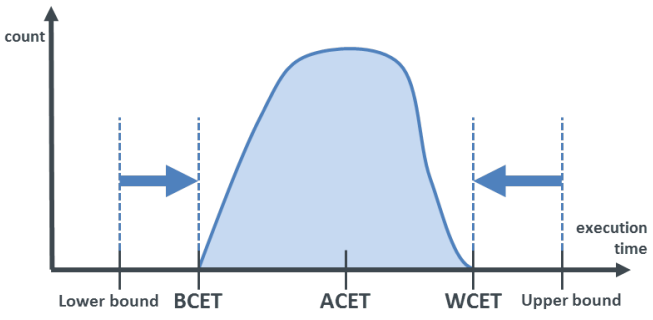

Figure 1: A distribution of execution times.

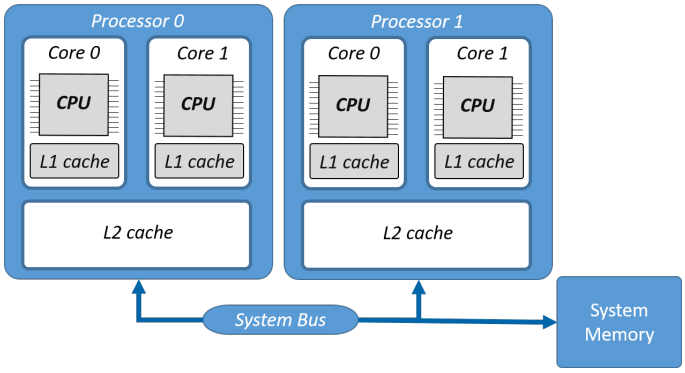

Figure 2: Shared cache: a dual-core, dual-processor system.

it will limit the benefits of having multiple cores, i.e. shared context between cores [11]. In [7] and [8], an approach to perform static analysis on shared caches is presented. Nonetheless, the assumptions made in the existing analyses are restrictive and the results are often very pessimistic or not even applicable directly to the current multi-core architectures.

A second approach is to develop new hardware architectures which support timing compositional analysis [12] [13]. These timing compositional architectures are decomposable into different subsystems whose timing distributions can be calculated straightforward. The challenge is to design an architecture with a performance comparable to that of current systems.

In summary, the current WCET analysis methods all fail as the soft- and hardware complexity increase, due to the increasing size of the system state space. A compositional approach offers an alternative that avoids the state space issue, but currently no viable compositional analysis method is available. We believe the solution for this shortcoming in the state of the art, lies in a hybrid approach.

3 Hybrid Methodology

Current WCET analysis methods have no acceptable solutions for complex systems, such as embedded multi-

core processors. The three methodologies discussed in section 2, all have their major drawbacks.

- The adaptation of the hard-/software to reduce the level of non-determinism for the **compositional analysis** (e.g. cache partitioning) are unacceptable because of their negative impact on the performance;
- The **measurement-based approach** requires too many measurements on complex systems to produce reliable WCET results;
- The **static code analysis** combined with abstract hardware models becomes useless as the size of the system state space makes the analysis intractable.

In order to overcome these shortcomings, we believe a hybrid approach to calculate the WCET of a task will provide the solution. The main problem can be described as '*the gap between a machine and a human being in solving problems*'. We, humans, think in algorithms and want to do estimates based on the structure of the algorithm. Machines on the other hand measure and calculate in bits and bytes. This hybrid approach will combine a time measurement-based approach at machine level with a static analysis approach at algorithm level (figure 3). The static analysis fits in well with the structured algorithmic thinking from the human. Additionally, the time measurements reduce the complexity for the machine. A schematic overview of this hybrid methodology is shown in figure 3.

The hybrid model splits the code of a set of tasks into so-called *basic blocks*. A basic block resembles a path trace of instructions which has only one entry point and one exit point [14]. The size of a block varies between the largest possible trace meeting the constraints of having a single in- and output point, and a single program instruction.

The challenge of this two-layer hybrid approach is tackling the computational complexity problems within the static analysis layer and the accuracy within the measurements-based layer. In other words, a proper balance needs to be found between the two layers of the hybrid model. This goal introduces four main challenges.

1. Finding a right block size in order to keep the complexity of the static analysis contained and to keep the accuracy of the conducted measurements, i.e. more blocks means more complex static analysis and fewer blocks means less accuracy in the measurement-based approach;
2. How to reduce the required amount of measurements while keeping a high accuracy based on the information from the static analysis layer. This can be achieved by reducing the measurement space using the static analysis information, e.g. excluding non-existing concurrencies;
3. Combining the information of the static and measurement-based layer to reduce the predicted upper bound by excluding irrelevant results. This will be accomplished by adding additional information to the measurements in order to use only the relevant subset of measurements. Only those results fitting the situation in the state of the static analysis will be used, e.g. not using correlation with start-up procedures in a shutdown phase;
4. Exploring the influence of the number of measurements conducted and their errors on the static analysis and the total WCET analysis. As a result, we will make the right trade-off between the measurement cost and the required accuracy depending on the specific application domain.

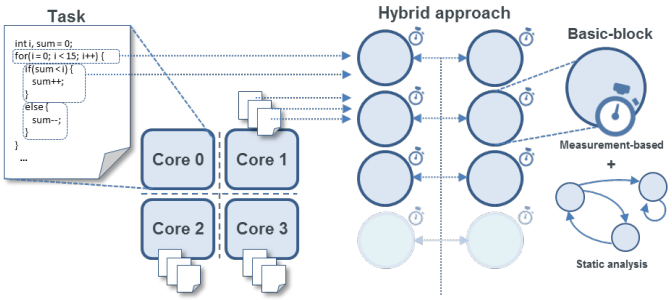

Figure 3: Hybrid cache analysis methodology.

Research on hybrid WCET methodologies has resulted in different analysis tools suites, such as TimeWeaver [15] and RapiTime [16]. These tools perform low level analysis on compiled code. As a result, they will only support a specific group of predefined target hardware and compilers which are examined by the tool developers.

Our goal is to create an analysis tool which performs and automates the analysis process based on a higher level, i.e. the code base. With this approach, we want to gain insight and find correlations between source code and the resulting WCET, so that embedded programmers can have continuous feedback on code changes during development and interpret the results on code level instead of binary code.

4 COBRA-HPA Framework

In the hybrid methodology discussed in section 3, the main idea is splitting the program code into basic blocks on which we can perform time measurements. In order to automate this block creation, we are creating a framework to help us verify our theories and finally allowing us to perform accurate WCET calculations.

The COBRA framework, *COde Behaviour fRAmework*, developed by the IDLab research group is an open source tool to perform various types of analysis to examine the performance and behaviour of code on different architectures. The framework allows

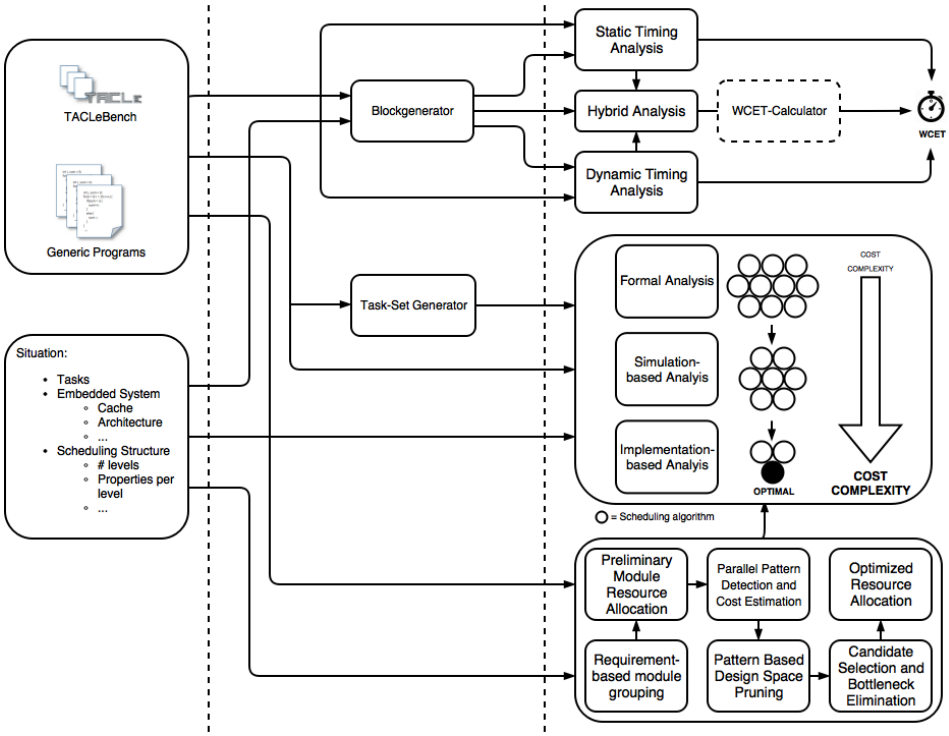

Figure 4: Schematic overview of the COBRA framework. Three sections can be distinguished by the dotted lines. The first section on the left contains the different sources of the input data. The middle section provides a set of tools which modify the input data of the first section and provide the output to the analysis tools in the third section. The last section on the right, contains the different analysis tools. From top to bottom: WCET analysis, optimal scheduling algorithm analysis and design pattern based performance optimisation for multi-core processors.

the developer to optimize the resource consumption on (currently) three main levels:

- **WCET analysis.** Different techniques exists to determine the worst-case execution time. The COBRA Framework implements both static and dynamic timing analysis to determine the timing behaviour.

- **Scheduler optimisation.** Information about the resource consumption, such as time (computing units) and energy, are used to optimise scheduling algorithms towards these resources for a specific application [17].

- **Design pattern based performance optimization for multi-core processors.** Applications can benefit from large speed-up on multi-core systems by parallelising their algorithms on different levels (e.g. code-based parallelism, instruction-level parallelism). However, finding and implementing the right design patterns to apply parallelism is a time-consuming task. The framework automates the process in finding the best design pattern for parallelism at the appropriate level.

An overview of the COBRA framework is given in figure 4. The framework itself consists of three parts which can be used separately. The first section provides input data from different sources, such as the TACLeBench benchmark suite [18], generic input programs or specific applications given a specific hardware configuration. The second section transforms the input to conform the framework defined description language with user-defined parameters used by the different tools. Because of this standardisation, new analysis tools only have to conform to this format in order to be supported by the COBRA framework. The last section includes all tools used to analyse and optimise the resource consumption for the three main levels discussed above.

In order to implement the hybrid analysis approach, we added an extension to the COBRA framework, the *Hybrid Program Analyser* or COBRA-HPA. This extension will parse any given C-source file and create a block tree on which we can perform algorithms. In the next step, the framework creates corresponding source files of each block which are used to run measurement-based analysis with. A schematic overview of the COBRA-HPA framework is shown in figure 5. The different modules in the tool are explained in the following sections of this section based on an example program.

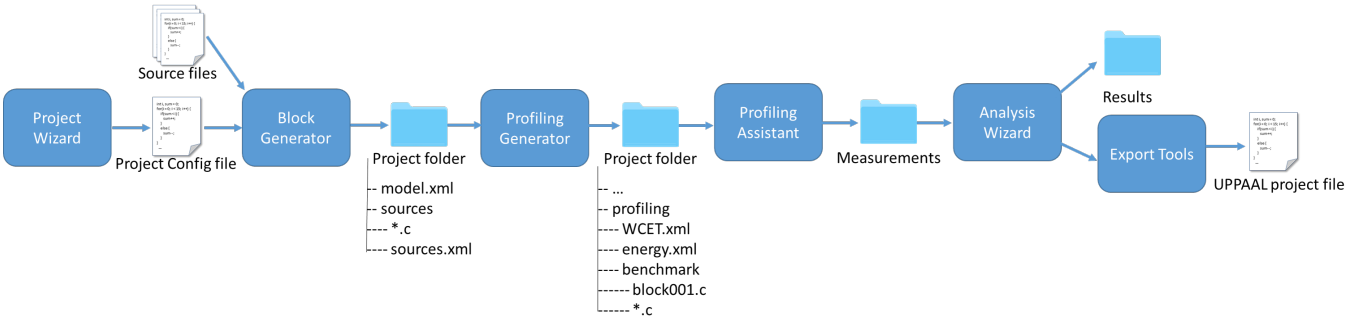

Figure 5: Schematic overview of the COBRA-HPA framework.

4.1 Project Wizard

The HPA toolchain, as shown in figure 5, perform its operation based on a *Project Configuration File* that will be used throughout the entire chain. This file contains the user-defined parameters characterising how the input source files need to be processed. The configuration files used in the toolchain are all formatted in XML-style.

The *Project Wizard* is a non-mandatory tool designed to give the researcher a user-friendly interface to define all parameters. Subsequently, the tool generates a configuration file conform to the requirements of the framework.

4.2 Block Generator

The main module of the HPA toolchain is the *Block Generator*. This tool allows us to generate a block representation of the input application according to the hybrid methodology discussed in section 3. In order to generate a block model, the *Block Generator* needs two different input sets as illustrated in figure 5, namely a project configuration file and the source files on which the analysis needs to be applied. After completion, the tool will create a project folder with the resulting block models. The project folder will contain a standardised model file which encodes the generated block models and the original source files in order to cross-reference each block to its original code segment. This project folder can then be used transparently by the other tools.

The *Block Generator* itself consists out of two main components, i.e. parsing of the source files and generating a block representation model of these files. The following subsections will explain these components in more explicit details.

4.2.1 File Parsing

The first stage in creating a block tree is parsing the code file. In this stage we need to interpret the source code to understand the structure of the code. In order to perform this task, we have chosen for the ANTLR v4 framework. ANTLR, *ANother Tool for Language Recognition*, is an open-source tool which can parse a text file according to a given grammar file [19] [20].

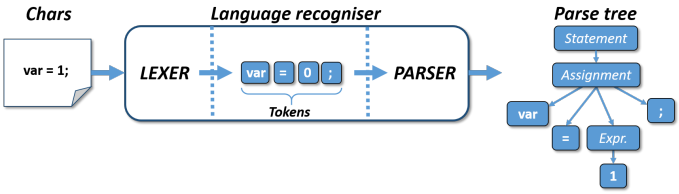

Figure 6: Language recogniser in ANTLR.

The parsing process is done by two components, the *lexer* and *parser* as shown in figure 6. The lexer splits the character stream into a series of meaningful tokens as defined in the provided grammar file [19]. These tokens are fed to the parser unit. The parser will then try to identify the parts of 'speech' according to syntactic rules in the grammar file [19]. The final result is a parse tree with the different tokens on its leaves and all rules on the intermediate nodes which were applied for each token [19].

A major advantage of the ANTLR framework is its language independence which allows us to switch to another programming language simply by loading the corresponding grammar file. Additionally, the automatic generated listener class enables us to walk through the tree and facilitates the integration of the ANTLR framework with our own.

In this first version of the COBRA-HPA extension, we implemented a file parsing unit for source files in the C-language. In order to generate this parsing unit, we need to compile an ANTLR-grammar file with the ANTLR-tool. For this we used an existing grammar from the ANTLR repository [20] instead of creating our own file from scratch. The syntax in this template grammar is conform to the C11 specification. However, some modifications were made to the grammar description to support compiler and flow-facts flags used in the test code from the TACLeBench and to facilitate the generation of blocks in the next stage. For example, we created a distinction between the true and false body of an if-statement, so we are able to couple each selection body to the corresponding statement result, i.e. true or false.

4.2.2 Block Generation

The second step in the generation of the block tree is walking the generated parse tree from the file parser. As previously mentioned, ANTLR v4 implements a visitor pattern with the automatic generated listener class [19]. The pointer object called the walker starts at the root of the tree. It will then travel from a node to the next child node in order, i.e. left to right. This walk pattern is also referred to as a depth-first search [19]. Each intermediate node contains a rule that was applied to a token. When the walker enters or leaves a node, it will notify the framework and trigger a listener event. In order to make the connection between the ANTLR framework and the COBRA-HPA framework, we have subclassed the empty listener class and overwritten the methods of interest. For example, when entering an iteration statement a message will notify the block generator that an iteration statement is detected. All the following statements will be contained inside the body of the created iteration block. The walker will publish an exit message upon leaving the iteration statement to tell the block generator that the block is completed and thus no further actions are required for this particular block.

The block tree used in the COBRA-HPA framework is generated on-the-fly when the parse tree is traversed by the walker. The result is a tree structure which is built out of blocks. Each block initially contains one instruction or statement of the source code. The blocks are arranged from left to right to their sequential order in the code. Instructions that reside inside a statement body are added as child '*blocks*' to the parent block. These instructions are characterised by their indentation in most programming languages.

In the first prototype, we created seven types of blocks in which we categorise each instruction. These blocks as shown in the class diagram in figure 7 are the following:

- Program Block: root block which defines an entire source file with all of its methods.
- Method Block: subcomponents of the program block that symbolises one program method.
- Jump Block: groups all instructions apart from iteration/selection instructions which will break the program flow by changing the instruction pointer to another program instruction, e.g. return, goto and break statements.
- Iteration Block: instructions that results in repeating a section of code depending on a stated condition, e.g. for, while and do ... while statements.
- Selection Block: includes those instructions where multiple program traces branch off the original trace. The given condition decides which trace will be followed, e.g. if ... else and switch statements.

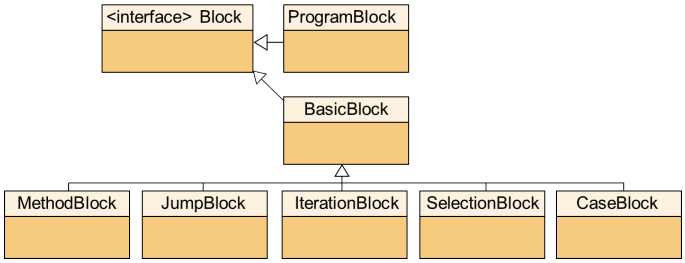

Figure 7: Class diagram of the COBRA-HPA block types.

- Case Block: these blocks indicate the different program traces originating from a selection statement. Case blocks hold the result for which the selection statement must satisfy in order to follow the underlying trace.
- Basic Block: contains all remaining instructions that cannot alter the program flow.

In figure 8, a block tree is rendered from the code of listing 1 to illustrate the structure of a block tree.

Listing 1: Sample program as an example input file for the COBRA-HPA framework. Source is written in the C-language.

```
int main(int argc, const char* argv) {
 int var1, var2;
 int array[10];

 if(argc == 1)
  var2 = atoi(argv);
 else
  var2 = 1;

 for(var1 = 0; var1 < 10; var1++) {
  array[var1] = var1 * var2;

  if(array[var1] > 100)
   break;
 }

 return 0;
}
```

The attentive reader will notice that the first basic block in figure 8 contains not one but two instructions. This aggregation of two blocks is the result of another implemented feature in the COBRA-HPA framework. This feature is called the *Block Reduction Rule* system.

In section 3, we discussed the importance of block size in our hybrid approach. We need to find a perfect balance to keep the analysis reliable and computable. The default block tree generated by the block generator will always contain one instruction for each block. In order to change the block size, we need to group the smaller blocks into bigger blocks. Therefore, the framework has the *Block Reduction Rule* system based on rules which allow the user to reduce the number of blocks by creating

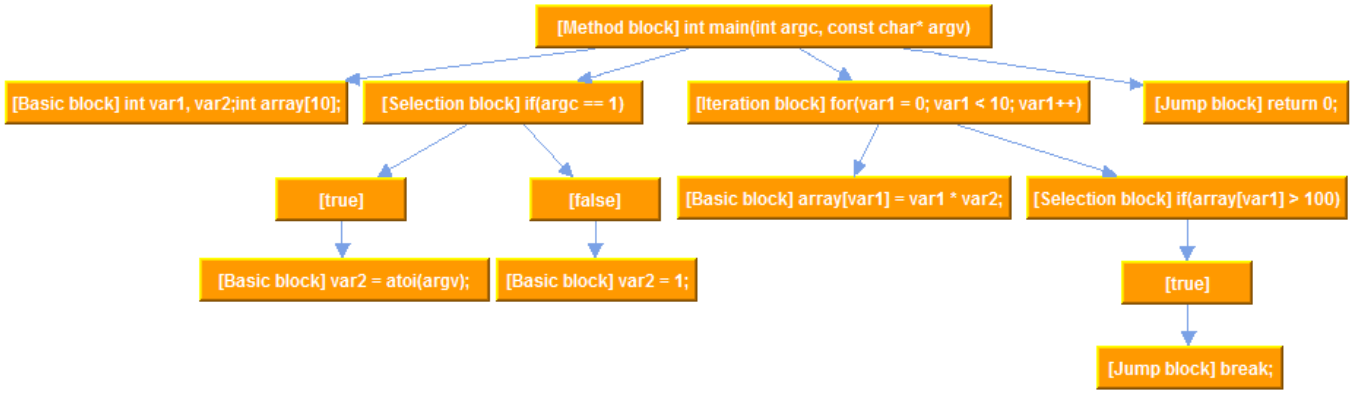

Figure 8: Block tree with COBRA-HPA of the sample code in listing 1.

bigger blocks. When the block generator is finished, the operator has the option to apply a set of rules on the model. The current version of the system has two rules implemented, i.e. basic block reduction and abstraction reduction. The first rule searches for successive basic blocks and replace those by one bigger basic block. The second rule groups all blocks starting from a user-defined depth and thus creating an abstraction from the lower level blocks. In the future, additional rules can be added to the system when more optimal reduction solutions are discovered with better key performance indicators (KPI) in respect to the resulting block sizes, e.g. computational complexity of the static analysis, WCET upper bound margin, etc.

4.3 Profiling Tools

The next step after creating a block model from a code file is profiling each block on the target platform. In the context of this paper, it means performing time measurements on the blocks in order to acquire the WCET. The COBRA-HPA framework provides supporting tools to simplify the profiling process and reduce the overall effort. The profiling section consists out of two major parts: Profiling Generator and Profiling Assistant.

4.3.1 Profiling Generator

The *Profiling Generator* is used in the preparatory stage of the block profiling. During this step, the tool generates the required code files from the block model. These code files are generated according to a syntax template which describes the correct formatting in order to create compilable code files for the profiling. Each generated file will include a main method to make the source file executable and a test bench method with the actual instructions of a block. Furthermore, interfaces are added for each profiling stage which allows the *Profiling Assistant* to implement platform specific code into the test bench, e.g. boot-up code, test initialisation, timer configuration, etc.

An additional challenge in creating a profiling test bench of a block is to identify the code dependencies and declarations as the code of a block is only a part of a larger application. Things get even more complicated when code dependencies are across multiple files. For example, when a variable is declared outside the scope of a block the *Profiling Generator* has to identify the missing type declaration first. Next, it has to find the appropriate type in the provided application code. Last, the variable declaration needs to be added before the test bench code so that the declaration will not interfere with the measurements of the block itself. Therefore, these declarations are placed in a dedicated function which get called before the test bench execution.

As the dependencies are declared, we need to initialise the input variables of the blocks. The goal is to generate input sets which include all possible input combinations during code execution in order to cover all cases. Furthermore, all non-occurring input sets need to be excluded to prevent too pessimistic results. Different strategies exist to generate valuable input sets to perform representative measurements [21] [22] [23]. The *Profiling Generator* performs three stages to derive values for the input sets. The first step is to perform a static analysis on the complete program to derive constant values or the state space of variables if possible. For example, the state space of an input variable i of a block is in the first place equal to the size of the data type. However, if we consider the context of the entire program this variable i can be defined as an iterator over a fixed interval in the parent block. This will eventually result in a significantly smaller state space of i.

The second step is to generate input sets for input variables which are annotated by the user. This step is optional, but is provided if the state space could not be derived in the first step. This brings the opportunity to refine the final results if the exact state space is known in advance, e.g. a temperature reading in the range of -30°C and 60°C.

The last step is creating the actual input sets for the measurements. The found state spaces of the previous steps are used first to parse the input variables file. Even a handful of input variables with a limited state space can eventually lead to an immense list of test that becomes impractically to test exhaustively. Therefore,

a random set creator allows to generate input sets according to the needs of the user. The size of the test set is adjustable through a custom configuration file. The variables which state spaces are undefined until this point, will receive a state space that is defined in the configuration file for each data type.

A last important feature which the *Profiling Generator* takes care of is gathering all external import and header dependencies. A list of these imports used in a basic block is generated during the static analysis of the corresponding code file and added to the test bench. At this point, we generated test bench code files for each block. These test bench files are used in the next stage to perform the actual measurements on the target hardware platform.

4.3.2 Profiling Assistant

The next tool in the profiling chain is the *Profiling Assistant*. As the name of the tool already reveals, its purpose is to assist with the actual profiling of the hardware platform. Therefore, it provides an interface with an extensive range of possibilities in order to support as many hardware platforms as possible.

Depending on the hardware, a different measurement approach needs to be taken. For example, dedicated debug pins on the hardware platform provide valuable inside into the processor inner workings which are mostly found on development boards. These pins allows us to perform accurate measurements. Nevertheless, if the debugging facilities are not available, other techniques will be applied to acquire an approximation.

The *Profiling Assistant* tries to select the most optimal, i.e. most accurate, methodology to perform the measurements on the target hardware. Initially, we presume the best-case where we know the full model of the processor architecture and full debugging functionality are available. If this is not the case, we will continue to fall back to less feasible approaches resulting in a more overestimated upper bound. E.g. on the ATMEL AT90CAN128 development board, we have full debugging access and sufficient input pins to provide test data. For this platform, we perform the profiling using an external device to keep the influence of the measurement to a minimum in order to reduce the probe effect on the hardware [24]. The external device is an FPGA with custom firmware which allows us on the one hand, to count the exact clock cycles to execute each block on the AT90CAN128 board with low interference. The FPGA uses the CPU clock of the micro-controller for counting the elapsed clock cycles. On the other hand, it delivers the input for each measurement as such that the flash memory of the ATMEL board is relieved from large datasets. As a result, we are not limited by the flash memory size when generating our input sets.

However, it is not always possible to use the previously mentioned procedure, as it is the case for the OCTA-Mini developed by IDLab. The prototype board is equipped with an ARM Cortex-M3. The board itself has no full debugging capabilities or as many IO-pins compared to the ATMEL development board. Therefore, a lower tier profiling approach is applied. The timing measurement is still performed by an external measurement device. Nevertheless, the input sets are now embedded onto the flash memory. As a result, multiple smaller tests are required to perform the analysis.

4.4 Analysis Wizard

By performing the measurements on the target hardware, we have completed the first part of the hybrid analysis, i.e. measurement-based analysis layer. The next step consists of evaluating the acquired results from all blocks and statically combining them according to their interactions in the control flow graph. As we are interested in the WCET of the application, we need to find the path with the longest execution time. The *Analysis Wizard* assists in estimating this path and its approximate WCET.

Firstly, the block interactions in the program are modelled with the program flow analysis. The consecutive execution of the blocks is representable as a flow through the program. In this phase, we need to identify all possible traces, or paths of the application. The flow analysis provides information, such as iteration counts, dependencies between selection statements, function calls, etc. in order to characterise the different paths. Therefore, these hints are called *flow facts*, as they acquire insight of the control flow graph of the program [14]. Flow facts can be derived implicitly from the program structure and its semantics, or by annotating the source code [14] [25]. The latter is required for flow facts which cannot be determined statically, such as recursion depths, dynamic calls and jumps, infeasible paths, loop iteration counts, etc. It is important to note that the user annotations have to consider all possible input sets in the system in order to obtain sound results.

Secondly, we need to calculate the WCET path in order to obtain the upper bound of the execution time. This is achieved by combining the measurements results from the previous tools and the flow facts of the program flow analysis. Currently, three major techniques are commonly used to compute the upper bound, i.e. *path-based* [26], *tree-based* [27] and *Implicit Path Enumeration Technique* (IPET) [28].

The first computation method is path-based calculation. A control flow graph of the program is used to search for the longest path as it will reflect the WCET. An example of a path-based approach is given in section 4.5.1. A significant bottleneck of this methodology is the computational complexity which rises exponentially with the number of control flow branches, as each possible patch has to be considered [14] [26].

The second method involves tree-based calculations. An estimate of the WCET is created by reverse traversing a syntax tree of the program. This

methodology is direct applicable to the block tree model that is used by this toolchain, as described in section 4.2.2. The leaf nodes of the tree contain basic blocks with larger code bases, which are connected by intermediate nodes that are controlling the program execution flow, e.g. iterations, jumps, etc. The execution time is estimated by combining the measurement results at the bottom according to the rule of the common parent node when traversing to the top. This method is computational feasible to perform, however it is not able to consider dependencies between statements [14] [25] [27].

The final method is the IPET technique. Finding the worst-case path is approached by transforming the control flow graph with basic blocks into a system of linear constraints. Each block and edge has a traversal counter and a cost (time). The given constraints can be solved using *Integer Linear Programming* (ILP) [28]. The next step is a matter of solving a maximisation problem. As a result, the sum of the traversal counter and the cost needs to be as high as possible in order to find the worst-case. The outcome will be the worst-case count for each block and edge. A big advantage of this approach is the ability to incorporate additional constraints into the equations to model flow facts and thus improving the upper bound precision [14] [28].

4.5 Export Tools

The last section of the framework collects all additional tools which enables the user to export the results to other formats or tools outside the COBRA framework for further research. At this moment, one export extension is already integrated in the framework, namely the UPPAAL Model Checker exporter [29].

4.5.1 Timed Automata with UPPAAL

In addition to the hybrid methodology discussed in section 3, the IDLab research group wants to explore other methodologies to determine the WCET. One of these studies is about determining the WCET with probabilistic analysis [30]. For this research, the theory of timed automata is applied. In order to support the creation of models for the experiments, we implemented a feature to build these automata automatically from source code.

A timed automaton is a finite-state machine extended with the notion of time [31]. The model contains clock variables which are in synchronisation with each other [31]. Therefore, the transitions in this model can also be guarded by comparing clock variables with numeric values [31].

For modelling the timed automaton, we use the model checker UPPAAL [29]. This toolbox developed by Uppsala University and Aalborg University is designed to verify systems which can be represented as a network of timed automata [31]. The purpose of using this tool is to model the program flow into a timed automaton. By implementing the UPPAAL model exporter, we are able to create timed automata.

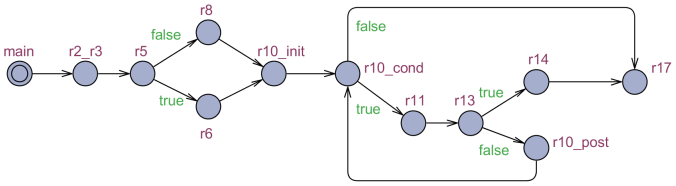

Figure 9: Timed Automaton in UPPAAL generated from the sample code in listing 1.

First of all, we start by creating a timed automaton data model. Therefore, we translate our hybrid block model to the syntax of timed automata. Each hybrid block is directly mapped to a node in the automaton. The next step is generating the necessary links between the nodes. These links are the transitions which represent the program traces that can be traversed during program execution. In order to create those links, we need to perform a program flow analysis to determine the different links or traces between the nodes. As previously discussed in subsection 4.2.2, we define seven types of blocks in our hybrid model. Each of these types refers to a specific structure inside the '*flow*' of the program such as iterating and selecting a program trace. To determine which links need to be created, we have to interpret the statements inside the blocks with the exception of generic basic blocks and case blocks. Each specific statement will eventually result in another program trace. For example, a while-loop will only iterate over the code as long the condition is true. Whereas a do ... while-loop will always execute the code at least once regardless of the result of the condition.

Second, the generated automaton model is exported into a XML project file for UPPAAL. The challenge in creating this file is to comply to the correct syntax defined by the UPPAAL framework and also to deliver a proper and clear layout of the model in the GUI based tool [29] [32].

In figure 9, an automaton is generated in UPPAAL for the example code given in listing 1. Each node in the model has a name corresponding to the line numbers in the original source file of the code included in that node. The associated code is included in the comments of each node. For a Case Block, i.e. true/false or cases in a switch instruction, the statement is added to the guard of the link that refers to the matching program trace.

5 Results

During the development of the COBRA-HPA framework, we are using benchmark programs to test the functionality and performance of our framework. The programs used as reference originate from the TACLeBench initiative. The TACLeBench is a benchmark project of the TACLe community to evaluate timing analysis tools and techniques in order to compare

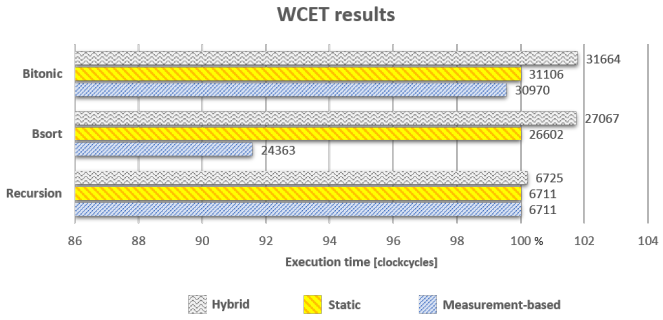

Figure 10: WCET analysis results comparison of three TACLeBench benchmarks.

their performances [18]. Each module is continuously tested with benchmark programs of the TACLeBench to verify the integrity and accuracy of the generated models. Apart from functional testing the toolchain, we need to evaluate the performance of the hybrid methodology and system.

As the toolchain is still in-development and does not yet implement all planned features, we run a few characteristic benchmarks to show the potential the framework would have when the full version is released. The following experiments were performed on the OCTA-Mini which is developed by IDLab. This board is a prototyping platform for embedded low-power wireless sensor IoT devices and is equipped with an ARM Cortex-M3.

Figure 10 shows the WCET results of the tests performed. For each test, we compared the hybrid analysis used in the framework with the static and measurement-based analysis. The bar graphs in figure 10 are normalised in percentages compared to the real WCET. With this representation, we can observe the deviation of the resulting WCETs with the actual one. Firstly, the hybrid results in the bar graph are obtained from the *COBRA-HPA* toolchain after the profiling step of the *Profiling Assistant*. Secondly, the static results are determined by finding the longest feasible path that results in the longest execution time. When the worst-execution path is found, a corresponding input set is composed which will trigger this path. The static analysis of the tests were calculated and verified manually. These results are therefore considered as the absolute WCET value. As a result, the static analysis bars in the graph are all equal to 100%. Finally, the measurement-based results are acquired by executing each program with random generated input sets for considerable long time. In the following subsections, we will discuss the significance and results of each test shown in figure 10.

5.1 Bitonic

The bitonic benchmark is a sorting network based on the bitonic algorithm. This sorting algorithm starts dividing the input set into pairs which it sorts alternating ascending and descending. Next, it groups consecutive blocks into bigger blocks, sorts them according to the desired order and repeats the process until the entire input set is sorted.

This benchmark in the TACLeBench has an input array of 32 elements. For this test, we made a small change to the input initialisation method as it is implemented in the benchmark suite. Instead of using the same input set for each run, we assume that each element of the array can vary in the interval [0,31].

As we compare the results in figure 10, we notice that the measurement-based approach makes an underestimation of the WCET by almost 0.5%. This means that the worst-case input was not tested. However, if we want to exhaustively test the benchmark with every possible input set, we would have 32^{32} different valid sets. When we consider the average execution time on the ARM Cortex-M3 with a clock frequency of 48 MHz, it would take over $3 * 10^{37}$ years to measure every input set! This approach is impossible to perform. Therefore, a random input generator delivers the required input data. Nevertheless, the measurement-based approach reached pretty close to the WCET bound for only measuring a fraction of all input sets ($< 0.01\%$).

The hybrid analysis on the other hand, has an overestimation of nearly 2%. However, an overestimation of the WCET is acceptable in contrast to an underestimation. The higher boundary is due to the current basic approach of the COBRA-HPA tool. Each basic block is measured independently from one another. The worst result of each block is then used in the static analysis. However, a worst-case path in one block should not necessarily be followed with the worst-case path of another block when a worst-case input set is given. Nevertheless, changing the block sizes presents the possibility to counter the effect when the upper bound would become too high. Further research in optimal block sizes is required as discussed in section 3.

When comparing the profiling effort of the hybrid method with the measurement-based approach, a tremendous improvement is noticed. The block with the largest input set has 'only' 1024 combinations which is equal to a complete profiling cycle of 1 s. A complete hybrid analysis took just a few minutes in total compared to the measurement-based approach.

The static analysis was performed manually by determining an input set which would result in the WCET of the program. At this point, it was rather straightforward to find this set for the algorithm because of its simplicity. However, when the programs are larger and more complex, it is less obvious. With the hybrid method, we are able to create a higher level abstraction of the program which simplifies the static analysis in exchange for accuracy of the upper bound.

5.2 Bubble Sort

The bsort benchmark is a bubblesort algorithm. This sorting algorithm iterates $n-1$ times over the input

array while it sorts the pointer element and its next neighbour to the requested order. The order will eventually propagate from the end to the begin of the array.

In the TACLeBench, this benchmark originally has an input array of 100 elements. Similar to the previous test, we made small adjustments to the code. The size of the input array is reduced to 25 elements. This change allows us to perform the measurements with a high accuracy without an overflow of the timer counter, i.e. a timer resolution equal to the CPU clock frequency. The second adjustment is to allow the input array to be randomised instead of using a fixed input set.

The results in figure 10 show the same trend as the bitonic test in section 5.1. The same problems arises with the measurement-based method. Now, an underestimation of more than 8% is recorded. The total input set combination is extremely high, which makes exhaustive testing not possible.

The same results for the hybrid methodology are observed as in section 5.1. An additional feature in the bubble sort benchmark is the use of nested iteration statements. These code constructions need to be handled with care when using the hybrid analysis. When the inner loops have variable loop bounds, it could lead to an exaggerated pessimistic upper bound. In order to support analysis tools, all TACLeBench programs are annotated with flow facts. For example, iteration statements are annotated with the minimum and maximum number of iterations. These loop boundaries hint the COBRA-HPA tool about flexible inner loops. If this is the case, the tool can tackle the problem in two ways. A first approach is to make an abstraction of the encapsulating loop so it will take the variable loop in consideration. The other approach is to lower the abstraction to the body of the loop and statically determine the total number of iterations if possible. In this experiment, the second approach was applied.

The annotations are an excellent feature to tell the analyser tools more specific details about the context and flow facts of a given program. However, when these annotations are not present in the code, the COBRA-HPA tool will try to determine the boundaries by itself. Nevertheless, when the boundaries could not be determined, the system is still able to adaptively change the block size creating a higher abstraction of the unknown loop boundary.

5.3 Recursion

The recursion benchmark calculates the Fibonacci sequence by invoking recursive method calls until the desired Fibonacci number is found. This benchmark only takes one input, namely the requested Fibonacci number. This test is a good test-case to evaluate the functionality of our tool on recursive algorithms.

As there is only one fixed input in the program that results in one fixed trace through the code, we can presume there will be only one final result for the execution time which equals the WCET. The results for the static and measurement-based analysis are therefore exactly the same as shown in figure 10.

The hybrid results only shows a small overhead of 0.2% compared to the static result. Determining the WCET of a recursive algorithm is a more complicated task for the static and hybrid analysis. The total state space can rapidly grow with each new recursive call which makes finding the longest feasible path hard or even impossible. In the case of the hybrid approach, our tool would consider the recursive call as a separate basic block. This results in an abstraction of the recursion itself by measuring all next recursive invocations. However, it could be desirable to go deeper into the recursion itself to increase the accuracy. This feature is not implemented yet, but it could be achieved by discovering basic mathematical series if present or alternatively by partially 'unrolling' the recursive loop.

6 Conclusion

In this paper, we discussed a hybrid approach to determine the WCET by combining two existing methodologies. This hybrid methodology creates an opportunity to merge the benefits of algorithm based thinking with a measurement based analysis.

As we strongly believe in the potential of this approach, we created the COBRA-HPA framework which allows us to split code into blocks according to the hybrid methodology. The resulting prototype is successfully capable of parsing C-source files and generating a corresponding hybrid block scheme. The toolchain provides an easy to use control flow to generated executable source code to profile the behaviour of the code, i.e. measuring the WCET of the hybrid blocks. Additionally, an extension on the framework makes it possible to translate an existing block model into a timed automaton for the model checker tool UPPAAL.

Finally, we profiled basic benchmarks from the TACLeBench with our prototype framework and compared the performance to the static and measurement-based analysis. We can conclude from these results that our approach reduces the number of measurements and thus the total analysis effort significantly compared to the measurement-based method. Additionally, the flexibility of the block sizes allows to create a higher level abstraction of the code. Therefore, providing the possibility to find a balance between the static complexity and the overall accuracy of the results.

In future work, we will continue on extending this prototype version of the COBRA-HPA framework with a more advanced static analysis layer. Additionally, more research on the influence of the block sizes will provide us the inside to create better algorithms for the *Block Generator*.

References

[1] Rob van der Meulen. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016, 2017.

[2] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[3] E. Bini and G. C. Buttazzo. The space of rate monotonic schedulability. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 169–178, 2002.

[4] Philip Axer et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 13(4):1–37, 2014.

[5] M Lv et al. A survey on cache analysis for real-time systems. *ACM Computing Surveys*, page 45, 2015.

[6] Jan Reineke. *Caches in WCET analysis*. PhD thesis, University of Saarlandes, 2008.

[7] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium*. IEEE, IEEE, 2008.

[8] Yun Liang et al. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012.

[9] Sebastian Hahn, Jan Reineke, and Reinhard Whilhem. Towards compositionality in execution time analysis - definition and challenges. In *The 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2013)*, 2013.

[10] Micaiah Chisholm et al. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. *Real-Time Systems Symposium, 2015 IEEE*, 2015.

[11] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. Multi-core cache hierarchies. *Synthesis Lectures on Computer Architecture*, 2011.

[12] M Schoeberl et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of System Architecture*, 2015.

[13] T Ungerer et al. parmerasa - multi-core execution of parallelised hard real-time applications supporting analysability. In *Euromicro Conference on Digital System Design*, volume 16, 2013.

[14] Paul Lokuciejewski and Marwedel Peter. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011.

[15] AbsInt. Timeweaver: Hybrid worst-case timing analysis, 2017.

[16] Rapita Systems Ltd. Rapitime - worst-case execution time (wcet) analysis for critical systems, 2017.

[17] Yorick De Bock, Sebastian Altmeyer, Jan Broeckhove, and Peter Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. In *Proceedings of the Embedded Operating Systems Workshop : EWiLi 2016*, pages 1–6, 2016.

[18] H Falk et al. Taclebench: a benchmark collection to support worst-case execution time research. *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET'16)*, 2016.

[19] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

[20] Terence Parr and Sam Harwell. Antlr project. GitHub repository, 2014.

[21] A. Ermedahl, J. Fredriksson, J. Gustafsson, and P. Altenbernd. Deriving the worst-case execution time input values. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 45–54, July 2009.

[22] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Design, Automation and Test in Europe*, pages 606–611 Vol. 1, March 2005.

[23] Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering Worst-Case Execution Times for Software Components. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based wcet analysis at the source level using object-level traces. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 54–63, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] Jong-In Lee, Su-Hyun Park, Ho-Jung Bang, Tai-Hyo Kim, and Sung-Deok Cha. A hybrid framework of worst-case execution time analysis for real-time embedded system software. In *2005 IEEE Aerospace Conference*, pages 1–10, March 2005.

[26] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times — a graph-based approach. *Real-Time Systems*, 13(1):67–91, Jul 1997.

[27] A. Betts and G. Bernat. Tree-based wcet analysis on instrumentation point graphs. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 8 pp.–, April 2006.

[28] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, November 1995.

[29] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. *Formal Methods for the Design of Real-Time Systems*, (3185), 2004.

[30] Haoxuan Li, Paul De Meulenaere, and Peter Hellinckx. Powerwindow: a multi-component taclebench benchmark for timing analysis. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, volume 1, pages 779–788. Springer Nature, oct 2016.

[31] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, chapter A Tutorial on UPPAAL, pages 200–236. Springer Berlin Heidelberg, 2004.

[32] Bart Koolmees. Validation of modeled behavior using uppaal. Final bachelor project, University of Technology Eindhoven, 2011.