Delft University of Technology
Master of Science Thesis in Embedded Systems

Parametric measurement-based WCET estimation for multiprocessor platforms

Caspar Treijtel
Supervised by Dr. Mitra Nasri

TU Delft Delft University of Technology

Parametric measurement-based WCET estimation for multiprocessor platforms

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Caspar Treijtel

November 3^{rd}, 2020

Author
Caspar Treijtel

Title
Parametric measurement-based WCET estimation for multiprocessor platforms

MSc Presentation Date
November 5^{th}, 2020

Graduation Committee

Prof. dr. Koen Langendoen (chairman)	Delft University of Technology
Dr. Mitra Nasri	Eindhoven University of Technology
Dr. Sorin Cotofana	Delft University of Technology

Abstract

Real-time systems are bound to timing constraints. These constraints are meant to ensure that the application exhibits predictable behavior by having bounded response times. The worst case execution time (WCET) is an important property of programs, which must be bounded to allow for a response time analysis of tasks. While estimation of the WCET is a difficult problem, modern commercial off-the-shelf processors with multiple processor cores make the WCET estimation problem even harder to solve, because of the existence of shared hardware resource contention among co-running tasks.

This work proposes a parametric WCET estimation tool, with which configurable and reproducible experiments can be created to investigate the co-runners' problem for specific task sets. The created tool runs on two hardware platforms, both featuring ARM Cortex quad core processors. Three different benchmarks suites are implemented in the tool, which can be configured to run in arbitrary combinations on the processor cores. These are a set of synthetic benchmarks meant to stress the memory system, a subset of the Mälardalen WCET benchmark suite and a subset of the San Diego vision benchmark suite.

The value of the tool is demonstrated through three sets of experiments. With these experiments, the effects of shared hardware resources are investigated in detail. We show that the experienced slowdown is highly dependent on multiple factors. First, a major factor is the sensitivity to the co-runners' effects of the task itself. Two extremes are investigated, from a task being insensitive to a task being highly sensitive. Another factor is the size of the input data, which is shown to be a major contributor to the experienced slowdown. Finally, we evaluate the delayed execution of co-running tasks, which in some cases has a significant effect on the experienced slowdown. When we consider a task's slowdown as a function of the delayed execution of its co-runners, the knowledge on this function's behavior for specific tasks provides an optimization strategy that could be used to mitigate the problem of shared hardware resource contention.

Preface

In our daily lives, computer systems are getting more and more ubiquitous. At the same time, their complexity is growing in a likewise fashion. This trend also holds true for embedded systems. When we speak of embedded systems, the user is normally not aware of any computer system, while he or she must still rely on the embedded system to be safe under all circumstances.

One aspect of a system's safety is the existence of any timing constraints that apply to applications running in a dynamically changing environment. Example applications are seen in the aviation or the automotive industry, which must have bounded response times to ensure that the embedded computer system's behavior remains both correct and on time.

A research problem that focuses on the timing of programs is called the worst case execution time (WCET) estimation problem. For modern off-the-shelf commercial processors, which are typically found in our daily lives, the WCET estimation problem is even more difficult. This is where this work hopes to add a contribution; by the creation of an analysis tool which can be used to evaluate configurable task sets.

This report describes the thesis project, which I have been doing for the last year. I would like to thank my supervisor for the project, dr. Mitra Nasri. First of all for providing me with the research topic and ideas, but also for her never ending dedication to her students.

My gratitude also goes out to my family and friends for bearing with me through all these years of studies. A special thanks goes to my girlfriend, who has unwillingly become knowledgeable in the real-time systems domain, because of my abundant real-time systems chatter and her proofreading of my texts. Finally, the most important thank you is reserved for my daughter, who has heard me say "I have to study" all too often.

Caspar Treijtel

Haarlem, The Netherlands
November 3^{rd}, 2020

Contents

Chapter 1

Introduction

The field of real-time systems is concerned with the correctness and efficiency of time-critical applications. In this context, correctness is defined as not only exhibiting functional correct behavior, also all timing constraints must be met. Typical examples of these applications can be found in the aviation or automotive domains. In real-time systems, tasks (system functionalities) require to interact with dynamic environments which must be monitored and acted upon, like the altitude of an aerial vehicle or the speed of a car. Since the environment is changing very fast, a task's response time must be bounded.

Typically, an application consists of multiple tasks, each of which has its own timing constraint denoted by a deadline. Some tasks may be more important than others, which is why tasks often have priorities over one another. Since each task needs the system's resources to do its work, its response time will depend on other tasks in the system that have a higher priority. Higher-priority tasks will take precedence over lower-priority tasks, and will only give up the system's resources once they are finished. They may even preempt lower-priority tasks, where the lower-priority task is suspended and will have to wait for the higher-priority task to finish.

To know if a given task set can successfully run in a time-critical system, the response time of each task in the system must be determined. The worst-case response times must always be smaller than the task's deadline. To determine the response time for each task, a response time analysis is performed, which is a theoretical analysis to obtain the worst-case response time of a task throughout the system's lifetime.

The corner stone of a response time analysis is notion of the *worst-case execution time* or WCET. Each task in the application executes software in the form of a compiled program binary. The program consists of a sequence of instructions that are executed by the system's processor. Each program, and therefore each task, has an associated WCET. It is defined as the longest time the task will ever spend on its execution. Normally, the true WCET is not known because the specific set of input data and sequence of executed instructions that lead to the WCET are generally unknown [34]. For most programs of reasonable complexity, the search space is simply too large. Instead, a task's WCET upper bound is used in the response time analysis. Without such an upper bound, no the response time analysis can be performed.

Determining the WCET upper bound has been an active area of research in

the past couple of decades [17], [6], [21], [24]. For single processor architectures, methods and tools exist that are able to analyze the program binary and produce a safe upper bound under strict assumptions [34]. The upper bound is said to be safe, if an analysis of the program binary proves the bound to be greater than any execution time of the program.

However, when dealing with multiprocessor architectures, methods and tools to determine the WCET's upper bound fall short [12], [17], [27] [31]. The added complexity of multiple processor cores makes a static analysis of the program binary very difficult. This especially holds true for systems where the multiprocessor is a *commercial off-the-shelf* (COTS) processor, because of the high levels of optimization combined with little disclosure on the inner workings of the processor. To be able to reliably use these powerful and economically attractive multiprocessors in a real-time systems context, the WCET estimation problem must be solved.

A key aspect of the WCET estimation problem on multiprocessor architectures is the problem of shared hardware resources. Multiple tasks simultaneously running in the multiprocessor, called co-running tasks or co-runners, create contention for these shared hardware resources. The contention created by co-runners can potentially add an unknown time to the execution times of tasks, in comparison to a scenario where the task runs in isolation. This makes the determination of the WCET upper bound very difficult. Prior work done by Bechtel and Yun has shown that tasks potentially suffer a slowdown of more than 300× when compared to the same tasks running in isolation [6]. A more recent study by Iorga et al., shows an even more pessimistic slowdown of nearly 400× as the worst-case result [21].

These results are alarming. The goal of these specific studies was to stress the system in order to find the highest possible slowdown due to contention for shared hardware resources. Not only are the slowdowns very high, since these studies were done with dynamic measurements, they still do not provide the system designer with a safe WCET upper bound. However, the derivation of a safe WCET upper bound is extremely hard [3], [34]. While for hard real-time systems the WCET upper bound must be safe, a recent survey of 120 industry practitioners in the field of real-time embedded systems shows that the industry heavily relies on measurement-based WCET estimation [1].

We postulate that the problem of shared hardware resource contention is very much dependent on the exact circumstances in which the task set runs. There are many factors that can influence the execution times of tasks, e.g. the specific set of other tasks that are running simultaneously, the operating system used, and the specific multiprocessor architecture on which the system runs.

That is why we propose a *parametric WCET estimation tool*, with which system designers are able to create reproducible experiments that are representative for their specific use cases; the task set and multiprocessor platform that constitute the system to be run in a real-time environment. This tool should offer the following functionality:

- The analysis of tasks in combination with freely chosen (arbitrary) co-running tasks;
- The variation of multiple hardware platform parameters that may influence execution times.

In addition, we identify an important shortcoming in the current state of the art of analysis tools. To the best of our knowledge, there has not yet been any study on the relative starting times of co-running tasks with respect to each other. Our work is the first to consider the effects of delayed execution of co-running tasks. Delayed execution of co-runners could potentially help mitigate the heavy slowdowns. We note that during the execution of a task, the usage of shared hardware resources will not always be uniformly distributed. We therefore propose to view the potential slowdown due to resource contention as a function of the alignment of the relative starting times of co-running tasks, by proposing a tool that allows for:

- The analysis of co-running tasks that vary in their relative starting times with respect to each other.

1.1 Research questions

In an attempt to provide answers to the aforementioned problem of resource contention among co-running tasks, we try to answer the following research questions.

- What are the factors that influence the WCET of tasks on a multiprocessor platform?
- How can we quantify these factors of influence?

In addition, the following hypotheses are tested to see if they hold true:

1. The distribution of the execution time of a task running in isolation is significantly affected by the existence of co-running tasks.
2. The distributions of the execution times of tasks that are concurrently running on multiple cores are significantly affected by the starting time offset of these tasks.

1.2 Contributions

Our work will contribute to available WCET analysis tools in the following ways. First, we propose to create the parametric WCET estimation tool, with which researchers and system designers can analyze specific task sets, by measurement of the execution times of tasks running in parallel. Second, we provide an experimental study that uses the proposed tool in an attempt to provide answers to the research questions listed above. These contributions are described further below.

1.2.1 Parametric WCET estimation tool

Our parametric WCET estimation tool allows for flexible configurations, with which experiments can be created and performed. The tool runs on two real hardware platforms that both have a COTS processor at their foundation:

- The ARM Cortex A53: an in-order execution quad core processor;

- The ARM Cortex A72: an out-of-order execution quad core processor.

These processors are found in the popular Raspberry Pi 3 model B and Raspberry Pi 4 model B computers, respectively. With these popular platforms, we believe to cover an representative part of available COTS processors, in which co-runners have been shown to create heavy slowdowns.

To evaluate our analysis tool, three types of benchmarks are selected. These are a subset of the Mälardalen benchmark suite [19], the San Diego Vision benchmark suite [32], and a set of synthetic benchmarks created with inspiration from prior work on timing predictability [6], [7].

1.2.2 Experimental study on co-runners' effects

Our second contribution is an experimental study that we perform using the proposed analysis tool. Specifically, we attempt to provide answers to the listed research questions and we present evidence for showing that the hypotheses listed above hold true, and explain the conditions under which they hold true.

Furthermore, our study indicates that the effect of co-runners is very much dependent on the specific conditions under which the task is being run. In our tool, we observe a worst-case slowdown of almost 150×, when compared to execution of the same task running in isolation. The slowdown results heavily depend on the specific parameters chosen in the estimation tool, such as the size of the input data of the task in question and the amount of memory that co-runners use.

Finally, an evaluation is presented of slowdown effects as a function of the size of the start time delay of the co-runners. This evaluation serves as a first investigation of a possible optimization strategy for shared hardware resource contention, using delayed execution of co-running tasks.

1.3 Organization

The organization of this report is as follows. In Chapter 2, an explanation of the co-runners' problem is presented in more detail, along with an overview of prior work on WCET research. Next, in Chapter 3 our problem statement is formulated, including a description of our system model and assumptions that apply to our solution. Chapter 4 presents our parametric WCET estimation tool. Chapter 5 provides the results of our experimental evaluation using the tool. Finally, Chapter 6 discusses our conclusions and directions for future work.

Chapter 2

Motivation

This chapter describes the problem of determining the WCET and how the contention for shared hardware resources makes this more difficult. The discussion explains our motivation for proposing our solution of the parametric WCET estimation tool.

2.1 The WCET estimation problem

Each task in the application exhibits a variation of execution times. The variation stems from multiple factors, such as differences in the input data or the external environment in which the application runs. In Figure 2.1, an example distribution of execution times is shown that captures the most important concepts related to the estimation of these execution times.

The worst possible execution time is called the *worst-case execution time* (WCET), which is defined as the longest time the task will ever spend on its execution. Likewise, the best possible execution time is called the *best-case*

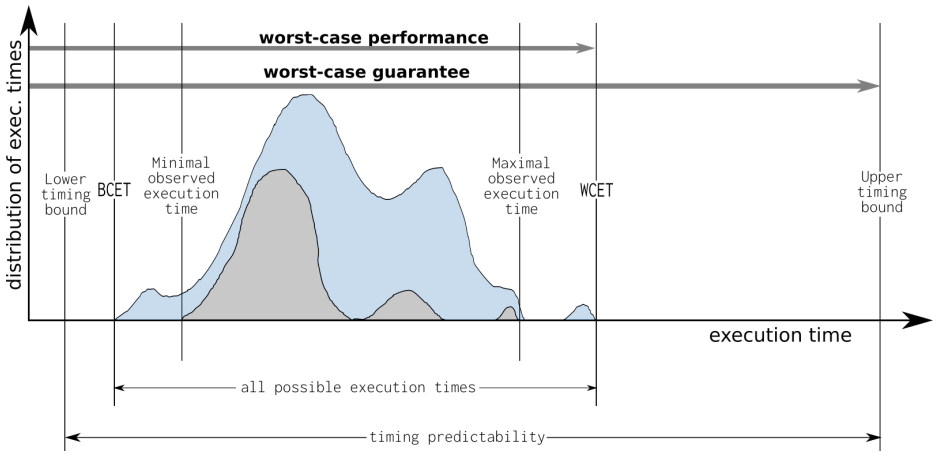


Figure 2.1: **The WCET estimation problem**
Dynamic measurement-based WCET estimation, actual BCET/WCET and lower/upper timing bounds by static analysis illustrated.
(Figure is derived from Wilhelm et al. [34])

execution time (BCET). In our work, we focus on the problem of estimation of the WCET.

Two strategies are used to determine the WCET. One strategy is to dynamically execute the task under study and measure its execution times. As will be clear from Figure 2.1, this strategy is not able to guarantee that the worst possible execution time has been seen. The other strategy is to examine the task through static analysis. Under some restrictions on the task's implementation, a static analysis can determine an upper bound to the WCET. The upper bound is said to be safe, if the static analysis proves that the upper bound is guaranteed to be greater than the true WCET.

For single processor architectures, methods and tools exist that allow for a static analysis of programs to provide safe upper bounds. A well known and extensive discussion is given by Wilhelm et al. [34]. The static analysis methods can produce safe upper bounds that are guaranteed to be greater than the actual WCET of the analyzed program. This does not mean that the computed upper bound is close to the actual WCET, in fact it can be quite pessimistic. One reason for the pessimism is the complexity of modern commercial off-the-shelf (COTS) processors, which are optimized for the average case performance. These modern processors are optimized for being having high performance most of the time, while under some circumstances, they can have far less performance. The possible sources of this timing variation are described in Section 2.2.

An interesting survey that has been done recently by Akesson et al., sheds light on the way industry is using methods and techniques from WCET research [1]. Although many respondents claim to be operating hard real-time systems, dynamic measurement-based WCET tools form the largest group of used tools. The survey's results support the claim that safe WCET estimation by static analysis is difficult. This notion must be kept in mind when reading Section 2.3 which describes prior work in the field of WCET research.

2.2 Sources of timing variation

The design of a processor is geared towards the principle of making the common case fast. This means that processors are optimized for the average case performance. How this contributes to the problem of WCET estimation is discussed in the following, where the processor pipeline and the memory hierarchy are explained.

2.2.1 Processor pipeline

Most processors implement a pipeline, in which the executed instructions are divided into separate chunks of work that can operate in parallel. The deeper the pipeline is, the greater the instruction level parallelism (ILP) can be. Since the instruction stages are smaller in a deeper pipeline, the processor can run at a higher frequency. There are trade-offs however, because the parallelism cannot always be exploited to its full extent. The processor pipeline can be stalled due to pipeline 'hazards' and memory stall cycles. Types of pipeline hazards are [20]:

- Structural hazards: there is a combination of instructions in the pipeline that are not supported by the lack of functional units or memory paths.

- Data hazards: the execution of an instruction is dependent on a result of a previous instruction that is not yet available.

- Control hazards: these are processor pipeline flushes that occur at branches and jumps. Branch prediction and speculative execution try to avoid these delays.

In Sprangle and Carmean [28], an example of a trade-off is described concerning the depth of the pipeline. The deep pipeline allows for a high processor frequency. However, the depth of the pipeline comes with a price, because a branch misprediction results in a stall of the pipeline. The deeper the pipeline is, the more instruction stages are flushed upon a branch misprediction.

The performance of a processor can be quantified by metrics, such as the number of cycles the processor needs to complete one instruction (*cycles per instruction* or *CPI*). If we consider a processor that can issue a single instruction at a time, the theoretical CPI is 1. However, hazards are bound to stall the processor pipeline at times, which will result in a higher CPI (less performance). The amount of stalls will depend on the specific program and the input data of the program.

In some cases, a reordering of instructions can potentially avoid structural or data hazards and keep pipeline stalls to a minimum, without effecting the functionality of the program. For an *in-order execution* processor, the instructions that are read from the program binary are executed in the exact same order as read. Therefore, any reordering of instructions can only be statically done by the compiler before runtime. Another type of processor is called the *out-of-order execution* processor. This type of processor allows for dynamic scheduling of instructions. This can lead to optimization of the running program on the hardware level. This may seem like a very modern processor-ability, but the underlying concept was first designed by Tomasulo for the System/360 architecture from IBM [30].

Some processors will have the possibility to issue multiple instructions at the same time, of which the *superscalar* processor is an example. For this type of processor a different metric is generally used, the *instructions per cycle* or *IPC*. The higher the IPC, the more performance the processor will have.

While the pipeline architecture with its optimizations can be a source of timing variation in its own right, this specific type of timing variation is not directly related to our problem with co-runners. However, indirectly the highly optimized processor with high IPC does contribute to the co-runners' problem in the following way.

The contribution to our problem lies within the fact that modern COTS processors typically are superscalar processors with the ability to run multiple threads and processor cores simultaneously. But since not all hardware resources in the processor are multiplied, there are still shared hardware resources for which contention among the tasks will be created. The highly optimized COTS processor has the ability to schedule multiple memory requests at the same time, while the memory system cannot handle that much requests simultaneously. The memory is an therefore important contributor to the co-runners' problem, as will be described in the next section.

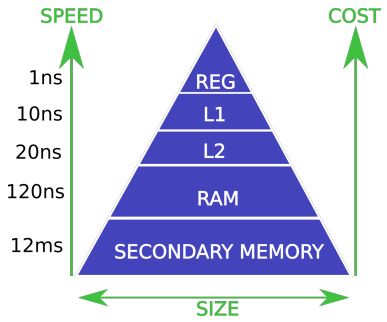


Figure 2.2: **Memory hierarchy**
The timings are indicative and show the order of magnitudes.
(Figure is derived from Patterson [26])

2.2.2 Memory hierarchy

Unfortunately, processor designers have to deal with the 'memory gap'. This expression refers to the fact that the speed of the processor has increased enormously over the years, while the speed of the memory has not. The result of the memory gap is that the peak performance of the processor is difficult to attain.

In an attempt to bridge the gap between the fast processor and the slow memory, a memory hierarchy is used. Processor designers *can* make use of very fast memory called cache memory, but this type of memory comes at very high cost and comes in very small (logical) sizes. Typically, the program and data will not fit into the small cache memory. To be able to store the program and data, larger memory is needed, which by definition is much slower. The spectrum of the types of memory is called the memory hierarchy and can be visualized in a pyramid structure, as shown in Figure 2.2.

Despite the fact that the program and data won't fit into the small cache memory, on average the processor can still achieve good performance. This is due to the principle of locality. Normally, the processor will be operating on a small *working set* of instructions and data, and will not randomly try to read any location from memory. The working set has a much better chance of fitting into the fast cache memory, which can lead to a high percentage of successful reads from the fast cache memory.[1] A successful read from the cache memory is called a *cache hit*. Whenever a memory location needs to be read and it is not found in the cache, a *cache miss* occurs. A cache miss will make the processor read the memory location from the slower memory, and copy its contents to the cache memory. If the cache memory was full, a less frequently used item will be removed from the cache memory to make room for the new item. The principle of locality equally applies to the larger (but slower) 'L2' cache, which is still much faster (but also much smaller) than the RAM.[2]

On startup of the program, its instruction and data must be read from the slowest type of memory (secondary memory). From there, the instructions and data will be copied to the faster memories, until they reach the processor's registers. From the processor's point of view, these initial reads will all be cache misses. This type of cache miss is called a compulsory cache miss, since it can

[1] Or writes to the fast cache memory.
[2] In some processor architectures more cache levels are present than shown in Figure 2.2.

never be avoided. Another type of cache miss is the capacity cache miss. This type of cache miss occurs when the cache is too small and data that was previously discarded must later be retrieved again. Finally, the conflict cache miss is a special type of cache miss that occurs in specific cache memory organizations, where memory locations are mapped onto cache sets. The conflict occurs when data from the cache is discarded due to conflicting memory locations that map to the same cache set. When the data must later be retrieved again, we speak of a conflict miss.

In the case of capacity and conflict cache misses, old data must be replaced by new data. The old data is said to be evicted from the cache. When the cache is organized in sets, the processor must choose a cache line that is to be replaced by the new memory block. A popular strategy to be used for choosing a cache line is called the *least recently used* (LRU) strategy. Like the name suggests, the cache line to be evicted is the one that has not been used for the longest time. An important aside to the LRU replacement policy, is that many processors implement a *pseudo* LRU replacement policy. This is an approximation to a true LRU replacement policy, to avoid an overly complex hardware implementation [26]. This distinction is important in the context of static WCET analyses, as will become clear in Section 2.3, where the cache analysis of programs is discussed.

For writes to memory, the cache and the main memory need to be kept in a consistent state. There are two strategies that are used in practice. The easiest strategy is called *write-through*. Here, the data is written to both the cache and main memory. A fast write buffer can be used before writing the data to main memory, in order to keep the pipeline from stalling. If writes are frequent however, or if they occur in bursts, the write buffer cannot keep up and the data must be written to main memory before the processor can continue.

A *write-back* strategy will write to the cache upon a cache hit, but will not (yet) update the main memory. This leaves the cache in a 'dirty' state, which is kept in an administration bit in the cache. When the cached data must be replaced by new data, the main memory must be made consistent before the new data can be written to the cache. This implies that a read miss can lead to a write to main memory. The dirty cache must first be written to main memory before it can be overwritten by the new data.

In the write-back strategy, a write buffer can also be used to minimize latency. When a read miss occurs, the dirty cache is first copied to the write buffer after which the memory can be read, filling the cache with the new data. Only then the main memory is updated by the write buffer.

The strategy chosen for writing to the cache has consequences for a static analysis of the task's WCET. This is further described in the next section, which contains an overview of research on WCET estimation.

2.3 Related work

In this section prior work is discussed that is related to our own work. First a discussion of WCET estimation for single processor architectures is given. Then we focus on multiprocessor architectures.

2.3.1 Single processor architectures

In the following, a subset of prior work on WCET estimation for single processor architectures is described. The work is classified according to the type of analysis, either static or dynamic.

We start this section with a remark on the types of WCET analysis. A common view of WCET analyses, is that static analysis methods will lead to safe bounds, while dynamic measurement based techniques will lead to unsafe bounds. This view is however over-simplified, and even called naive by Altmeyer et al. [3]. The authors state that all WCET analysis methods have potential sources of errors. For static analyses, the complexity of the hardware and the sometimes lack of knowledge on its inner workings is a problem. Analysis of the program source code is difficult because of compiler optimization. For dynamic measurements, the problem lies in the fact that the technique relies on high quality test data, which we cannot always be guaranteed to have.

Static analysis

As described in Chapter 2, the memory hierarchy is an important factor that contributes to the execution time of a task. Therefore, a lot of prior work is focused on the cache. The goal of these studies is to produce a WCET upper bound that is more precise, when compared to an analysis that assumes that all memory requests are cache misses and thus go through the main memory. In this section, a small subset of prior work on cache analysis is described. We note that a more extensive survey is to be found in Lv et al. [23].

To be able to reason about a program's cache behavior without actually running it, the analysis must abstract away from concrete values. The amount of possible input data value combinations is too large to analyze in general. A technique that is used to reach the level of abstraction is called abstract interpretation [10].

The first work to consider a more precise WCET by including a cache analysis was done by Alt et al. [2]. Abstract semantics are applied to the program, to be able to predict the contents of the cache throughout the program's flow. With this information, the analysis allows for replacing some memory requests with a cache miss or cache hit event. This sharpens the otherwise pessimistic WCET calculation, which assumes that all memory access go through main memory.

The cache analysis must have knowledge on the replacement policy implemented by the cache architecture. With the information on the replacement of each abstract cache line, a lower and upper bound of the cache line's age can be determined. An upper bound on the age of a cache line can lead to a certain cache hit and a lower bound can lead to a certain cache miss. The upper and lower bound derivation is also known as a must and may analysis, respectively.

The contribution of the work of Alt et al. is a generic cache analysis with the cache replacement policy as a parameter. A similar work was published by Ferdinand and Wilhelm [16] in a journal article, extended with experimental results of a set of test programs.

In the context of real-time systems, the specific type of cache implementation can have consequences. Take for example the write-back cache, the alternative to the write-through cache. The write-back cache is a popular choice made by multiprocessor designers, since this type of cache requires less memory band-

width [20]. However, for a static analysis of the program, the write-back cache's behavior is harder to analyze because of writes that may or may not happen some time in the future.

Blaß et al. [8] have extended the earlier work on cache analysis techniques, to further improve the precision of the WCET upper bound for write back caches. Blaß et al. state that instead of focusing on cache evictions, they focus on stores to the cache. Since a store to an already dirty cache line does not lead to actual write back to memory, their analysis is said to improve the precision.

We note that although the static analysis should be able to provide a safe upper bound to the WCET, the approach is not safe for many processor architectures with timing anomalies [34, sec. 2.1.3].

Dynamic analysis

In this section, prior work on dynamic analysis techniques is discussed. Most of the prior work follows a hybrid approach, where measurements are combined with static analysis of the program. This type of hybrid analysis is also known as *measurement-based timing analysis* (MBTA), or a variant thereof called *measurement-based probabilistic timing analysis* (MBPTA).

Measurement-based timing analysis (MBTA)
The work of Deverge and Puaut [14] is based on structural testing methods that are able to generate input data, which are used to exhaustively test all possible program paths. According to Deverge and Puaut, their method "would" produce safe and precise bounds. However, an important assumption that Deverge and Puaut make, is that measurements of executions of the same program path will always yield the same results, regardless of the input values. For this assumption to hold true, the processor architecture must be controlled.

In Wenzel et al. [33], the program to be analyzed is divided into segments. The segments are individually analyzed and measured, to be composed into a final estimation of the WCET. The source code of the program under study is instrumented, i.e. it is enriched with annotations. Wenzel et al. state that their method allows for derivation of safe WCET estimations, even when dealing with complex hardware. However, loops are not supported.

Measurement-based probabilistic timing analysis (MBPTA)
A variant of the MBTA technique is MBPTA, which is based on probability theory. Instead of reasoning about the WCET, in MBPTA the bound is called a probabilistic WCET (pWCET). The reasoning behind the concept of a pWCET is that a violation of a bound is viewed as a system failure. The application assumes acceptance thresholds to failure probabilities, and when the MBPTA analysis provides bounds with failure probabilities that fall beneath the thresholds, the application can assume the bounds are safe.

Cucu-Grosjean et al. [11] propose to use the MBPTA technique. Their approach is based on Extreme Value Theory (EVT). The authors state that their EVT-MBPTA technique is able to provide tight pWCET estimates.

2.3.2 Multiprocessor architectures

This section contains a selection of prior work that focuses on the problem of WCET estimation for multiprocessor architectures. For completeness, the

reader is referred to a survey done by Maiza et al. [24].

Several approaches to attacking the problem of resource contention have been proposed. In the work of Andersson et al. [4], a formal model of co-runner dependent tasks is presented, with which a schedulability test can be performed. The model is expressive enough to assign a WCET to a task as a function of the set of co-runners that execute at the same time. The authors' contribution does not focus on estimation of the WCET. Instead, Andersson et al. put co-runner dependent WCET's as parameters in the model, with which schedulability tests can be run that have polynomial time complexity.

Davis et al. [12] propose a multicore response time analysis framework, in which a time predictable architecture can be specified by a parameterized hardware configuration. The performance of specified, time predictable multicore systems is evaluated with the framework to guarantee time-predictable performance. The results are compared to a reference framework, which is designed for average case performance. Being able to guarantee safe bounds on a multicore systems is very attractive, however the assumption made by Davis et al. is that the application can run on their new processor architecture. In our work we assume that the application will run on a COTS processor.

Radojković et al. [27] also assume that the application runs on a COTS processor. They were one of the first to propose a methodology to dynamically quantify slowdowns due to shared resource contention in multi-threaded processors. Their methodology consists of a set of specialized synthetic benchmarks that are meant to stress shared hardware resources. Their study shows that measurement-based timing analysis cannot be directly extended from single threaded to multi-threaded architectures. Although their ideas are very similar to our proposed solution, some important differences can be identified. First of all, their synthetic benchmarks that are meant to stress the platform, are written in assembly code which are very specific to the architectures under study. Secondly, their methodology relies on running experiments on a full fledged operating system (Linux). The choice of Linux implies more work with respect to controlling the environment in which measurements are taken. Lastly, although their results do show that on some architectures co-runners cannot be run in a time-critical environment, newer studies on more modern processor architectures show far worse slowdowns.

One such study is has been done by Bechtel and Yun [6]. They were the first to show heavy slowdowns by a factor of more than 300×. Their experiments were done on several modern COTS processors, with the worst slowdown measured on the Raspberry Pi 3. Bechtel and Yun propose a mitigation solution to the co-runners problem, which is based on MemGuard [36]. MemGuard allows for the creation of a throttle for low priority tasks, to make sure that enough bandwidth is reserved for high priority tasks. An interesting result reported by Bechtel and Yun is the fact that in-order processor cores (like the ARM Cortex A53, running the Raspberry Pi 3) are also susceptible to severe slowdowns due to shared resource contention. Previous studies suggested that especially out-of-order processor cores were vulnerable to the co-runners problem.

An example is the study that has been done by Valsan et al. [31]. They have focused on a set of architectures with out-of-order processor cores. Valsan et al. also propose a mitigation solution to deal with co-runners. The MSHR register, which is part of the non-blocking LLC, is identified as the root cause of the shared resource contention. The proposed solution is a combination of

both system software and hardware modifications. Since hardware modification is not possible in a COTS processor, Valsan et al. evaluate their solution by implementing the hardware extension in a cycle accurate simulator. Despite the efficiency of their solution, we consider a hardware modification as a very difficult or even impossible undertaking. At least in the foreseeable future, where COTS processor design is still mostly proprietary and governed by a large commercial market wanting high performance processors at low cost.

A different mitigation strategy has been proposed by Xu et al. [35]. The authors allocate shared hardware resources to tasks, in an attempt to find an optimal trade-off between cache usage and memory bandwidth usage. Xu et al. note that when a task is allocated more cache, its memory bandwidth demand will be less (and vice versa). By grouping tasks with similar resource demand characteristics, tasks can fully utilize the assigned resources. An important part of the work of Xu et al. is finding an optimal resource allocation strategy. Tasks are empirically evaluated in isolation, where they are throttled in their resource usage to show their resource demand characteristics. The authors' goal is to find a near optimal processor utilization, by finding the maximum schedulability of given tasks under resource constraints.

A recent study that can be compared to our own work has been done by Iorga et al. [21]. Like in the work of Bechtel and Yun and in our own work, Iorga et al. also create synthetic benchmarks that are meant to stress the shared hardware resources, creating problems for the *program under test*. The authors do not propose a mitigation strategy. Their focus is on creating a methodology for WCET estimation of programs that run on a multiprocessor platform. The authors describe an extensive evaluation, in which the results of Bechtel and Yun are reproduced and even surpassed by their new synthetic benchmarks (*enemy programs*). The selling point of the authors' methodology is the fact that their synthetic benchmarks are *auto-tuned*. Instead of manually creating a highly specialized and effective enemy program, Iorga et al. rely on optimization techniques for automatically finding the worst possible enemy program. We recognize the attractiveness of their approach, but we note that in our work the co-runners problem is viewed from a different angle. Our aim is not only to try to approach the WCET in our estimations, but also to show that the relative starting times of co-runners can be of significant importance. Prior knowledge on timing variation as a function of tasks' starting times could be used by a scheduling algorithm in an attempt to optimize for processor utilization. This idea is described in the next chapter, where we define our problem statement and the system model that applies to our solution.

Chapter 3

Problem statement and system model

In this chapter, our problem statement is formulated and the system model is defined that forms the basis of our proposed solution.

3.1 Problem statement

As described in Chapter 1, our work is focused on WCET estimation on COTS multiprocessor architectures. When dealing with COTS multiprocessor architectures, we come into the realm of co-runners and the shared resource problems they create. We are left with the dynamic estimation of the WCET, as shown by the results of an industry survey [1]. Deriving a safe upper bound by means of a static WCET analysis is generally considered to be infeasible for COTS multiprocessor architectures, for reasons of complexity and lack of knowledge about the inner workings of commercial processors (see e.g. [12], [17], [27] and [31]).

A typical optimization of a modern COTS processor is the non-blocking cache. This optimization is identified as an important contributor to the shared hardware resource problem by Bechtel and Yun [6]. In the following section, the non-blocking cache is described.

Non-blocking cache

The non-blocking cache is an optimization that improves the processor's performance with respect to memory accesses. This type of cache will not block other instructions upon a cache miss, instead it will be able to service other memory accesses while data is being fetched from memory. As described in Hennessy and Patterson [20], this optimization is a feature of processors that have an out-of-order execution. However, the non-blocking cache is especially beneficial in any multiprocessor architecture, since multiple processor cores can issue memory requests simultaneously [6].

Naturally, there are limitations to the number of caches misses that the processor can handle simultaneously. The limitation comes from the fact that the processor has to remember outstanding memory requests. The implementation

of the non-blocking cache is based on a write buffer and an auxiliary register called the *miss information/status holding register* (MSHR) [22]. This is a register in which cache misses are recorded, until the data is read from memory. With the information in this register, the cache is able to service other requests.

As reported in Bechtel and Yun [6], in the non-blocking cache, both the write buffer and the MSHR will block the cache when either of them becomes full. The non-blocking cache is identified as a major source of the co-runners' problem, acting like a bottleneck when many memory requests are issued by multiple cores.

3.2 System model and assumptions

In this section our system model and underlying assumptions are described, which serve as the foundation for our solution. Our system model consists of a model of the processor and a task model.

Time-critical applications are classified according to the severity of failure to meet the timing constraints. A *hard* real-time system is a system in which deadline misses can lead to catastrophic conditions. In a *firm* or *soft* real-time system, a deadline miss will not be critical but the system's output or performance can be degraded. In this work, we assume that the application is classified as a firm real-time system.

3.2.1 Multiprocessor model

Our multiprocessor model consists of n processor cores. Each core has a local instruction cache and data cache. The first core, core 0, will always run the task that is being evaluated. The other cores are designated as 'co-runners', they may create problems for the task running on core 0. The processor model is graphically depicted in Figure 3.1.

3.2.2 Assumptions

Our task model consists of a set of at most m tasks and n processor cores, where $m \leqslant n$.

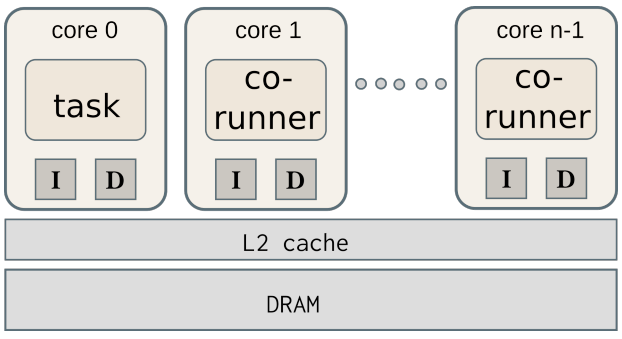


Figure 3.1: **Multiprocessor model**

Non-preemptive execution

Normally, the WCET applies to the longest execution time of the task when running in isolation. When a preemptive scheduling algorithm is applied to the task set, the analysis of the WCET must be extended with an analysis that takes into account the overhead of context switches, due to preemption of higher priority tasks. One type of overhead is called *cache-related preemption delays* (CRPD). The contents of the cache could be replaced by higher priority tasks upon preemption, incurring more execution time for the lower priority tasks.

In this work, non-preemptive execution is assumed and preemption overhead is therefore not taken into account.

Chapter 4

Parametric WCET estimation tool

In this chapter, our parametric WCET estimation tool is described. First our goal and requirements for the tool are presented, after which the specific functionality and prerequisites are described. Finally, this chapter contains an overview of the technical design that was made for the tool. Details of the implementation are described in Appendix A.

4.1 Goal and requirements

Our primary goal for the research project is to investigate the effect of co-runners for tasks that are run in a multiprocessor environment. We want to be able to create reproducible experiments for arbitrarily chosen programs or task sets. These tasks must be investigated when run in isolation and when run with numerous co-runner configurations. Also, we want to be able to synchronize all tasks running in the system and create delays for the co-runners. This way we can evaluate the effects of delayed execution of the co-runners.

In order to meet our goal, we propose to create the parametric WCET estimation tool. In Figure 4.1, a global view of the tool is shown with inputs and output. The inputs are **(1)** a set of tasks for each experiment that from the task set, **(2)** a set of parameters that may influence the co-runners' effect and **(3)** the platform on which the experiment should run. These parameters will be discussed in more detail in the next sections.

From our goal we can distill the main requirements for the tool. These are

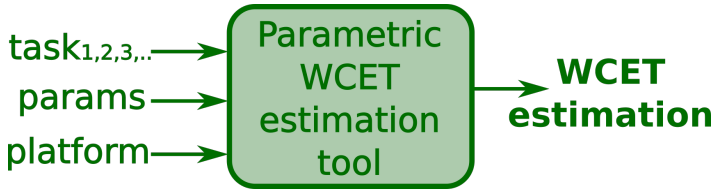


Figure 4.1: **Global view of the tool with main inputs and output**

Req nr	Requirement	Explanation
R1	Create experiments	User must have a user friendly interface to the tool with which multiple experiments can be created.
R2	Flexible configuration	Arbitrarily chosen tasks or benchmarks can be incorporated into the tool.
R3	Multiple experiments	The tool must be able to automatically read and execute multiple experiments.
R4	Multiple iterations	The experiments must be repeatable, i.e. the tool must be able to create multiple iterations for the same experiment.
R5	Parametric experiments	Parameters can be set in the tool that influence execution time, examples are MMU on/off, memory sizes, etc.
R6	Measure cycles	The execution time of the tasks that run on all cores must be measured, measurement unit is processor cycles.
R7	Measure events	The events of interest to the experiment must be monitored, example events are cache hits/misses, TLB hits/misses, etc.
R8	Synchronized execution	Co-runners must be run in sync with the task being evaluated.
R9	Delayed execution	Co-runners must be run with varying delayed execution, e.g. $1/10^{th}$ of baseline, $2/10^{th}$ of the baseline, etc.
R10	Reporting functionality	The tool must be able create log files and graphical reports containing information on the execution times.

Table 4.1: **Global requirements for the parametric WCET estimation tool**

listed in Table 4.1. In Section 4.2 these requirements are further described and translated into more detailed functionalities and prerequisites.

4.2 Functionality and prerequisites

This section describes the functionality and prerequisites that apply to our parametric WCET estimation tool. Also, a set of benchmarks is described that are used for evaluation of our tool. Appendix A contains more in-depth information on the specific details of its functionality and implementation.

R1 — Create experiments

The user must have a friendly interface to the tool, with which multiple experiments can be created. These experiments must also be **reproducible**, by which we mean that we and other researchers must be able to reproduce experiments under the same circumstances.

In terms of functionality, we have chosen a spreadsheet tool for the creation of experiments. Each experiment is created by defining its configuration and parameters row-wise.

R2 — Flexible configurations

We require our tool to be flexible in its configuration. That means that the user must be able to add new programs or benchmarks into the tool, for evaluation of their sensitivity to co-runners. This requirement does not lead to a new functionality, but leads to a prerequisite to the implementation; the new program or benchmark is to be added to the framework with little porting effort.[1]

R3 — Multiple experiments

The tool must be able to automatically read and execute multiple experiments, that are defined by the user in the spreadsheet. This contributes to the requirement of creating reproducible experiments without being labour intensive.

R4 — Multiple iterations

The experiments must be repeatable, i.e. the tool must be able to create multiple iterations for the same experiment. This translates into the functional requirement that the tool automatically runs multiple iterations, where in each iteration the same conditions are created.

R5 — Parametric experiments

Parameters can be set in the tool that influence execution time. The parameters that must be configurable in the tool are:

- **Benchmark configuration** — Specify the configuration of benchmark to processor core mapping.
- **MMU** — Turn the memory management unit (MMU) on or off. When the MMU is turned off, the L1 and L2 data caches cannot be used. This could have a significant impact on slowdown effects due to co-running tasks.
- **Screen** — Turn on the screen functionality on or off. This may also have an impact on the level of cache usage by cores and thus may influence slowdown effects.
- **Cache management** — To be able to run multiple iterations, the same conditions must be created for each iteration. For this the cache is managed. However, the user must be able to turn the cache management off.
- **Input size** — An important factor in the amount of slowdown could be the size of the input data that the tasks receive. With this parameter, the input data size can be controlled for each task.
- **Experiment label** — For each experiment, a user defined label can be created, which is used in the reporting functionality described below.

[1] We assume that programs or benchmarks to be included are written in C.

R6 — Measure cycles

The execution time of the tasks that run on all cores must be measured. In our tool the metric for the execution time is the number of cycles spent by the task.

R7 — Measure events

The events of interest to the experiment must be monitored, that are indicator for the performance of the processor cores. Example events are cache hits and cache misses, memory accesses, TLB hits and TLB misses, etc. In Appendix A the implemented events are listed.

R8 — Synchronized execution

The tool must be able to have the co-runners start synchronously with the task being evaluated. This means that they have an equal starting time.

R9 — Delayed execution

The tool must facilitate a varying delayed execution for the co-runners. The delayed execution will be one or more fractions of the baseline WCET. The baseline WCET is defined as the worst execution time when the task to evaluate is run in isolation.

R10 — Reporting functionality

The tool must be able create log files and graphical reports containing information on the execution times. Specifically, for each iteration, the following data is produced:

- Experiment label;
- Configuration of benchmark to core mapping;
- Benchmark name;
- Number of cores taking part in the experiment;
- Core number;
- Number of cycles spent;
- Co-runners' starting time offset (delayed execution);
- Iteration number;
- Event type en number of recorded events;

The produced log files must be created in the CSV format. They must be further analyzed for the creation of basic statistics of the experiment. These are the median, the maximum (WCET) and the standard deviation of all measured cycles during each iterations.

Additionally, a linear chart must be created from the data for each experiment, in which the number of cycles for each iteration is shown. This type of chart will allow for evaluation of the validity of the experiment, because all iteration

measurements should be independent of one another. If the data shows an increase or decrease with growing iteration number, the experiment may not be valid. A typical example is the 'warming up' of cache memory, where subsequent iterations show improved performance when compared to previous iterations.

4.2.1 Prerequisites

In addition to the described functionality, the following prerequisites apply to our tool. First of all, as described in Section 3.2.2, we assume a non-preemptive scheduling algorithm. That means that when a task (or a co-runner's task) starts, it must not be preempted. The system must therefore make sure that the tasks are not interrupted during execution.

Furthermore, the tool must be resilient against system failures. When for whatever reason an experiment goes wrong, the tool has to be able to recognize the failure and automatically repeat the same experiment.

Finally, we note that users of our tool should fall within our user target group, these are experienced computer users or software developers. Examples of typical users are researchers or research software engineers, with some knowledge and experience of the C programming language.

4.2.2 Benchmarks for evaluation of the tool

In this section, the benchmarks are described that are used for evaluation of the parametric WCET estimation tool. The benchmarks fall into three categories, these are synthetic benchmarks, benchmarks from the Mälardalen benchmarks suite and benchmarks from the San Diego vision benchmark suite.

Synthetic benchmarks

The synthetic benchmarks are created in this work. The purpose of these benchmarks is to stress the platform, by continuously reading and writing to memory. The following benchmarks are implemented, with inspiration from prior work [6], [7].

1. **linear array read** — continuously read from memory, by reading from an array. The memory locations are chosen to be 64 bytes apart on each iteration in the `for` loop, thereby 'optimally' stressing the cache, because of the 64 bytes cache line size in the chosen processor architectures;

2. **linear array write** — continuously write to memory, analogous to the linear array read described above;

3. **random array read** — continuously read from memory, but from randomly chosen memory locations. When the benchmark is being run, the random locations are already prepared beforehand, in an attempt to only stress the memory.

4. **random array write** — continuously write to memory, analogous to the random array read described above.

Mälardalen benchmark suite

The Mälardalen benchmark suite, created by the Mälardalen WCET research group, is a benchmark suite that was created to be able to compare WCET tools from different paradigms [19].

The benchmarks from the Mälardalen benchmark suite listed below have been ported to our tool:

1. `bsort100` — `bsort100` is a Bubblesort program. It tests basic loop constructs, integer comparisons and simple array handling by sorting integers.
2. `ns` — Search in a multi-dimensional array. Here, deep loop nesting is tested, 4 levels deep.
3. `matmult` — Matrix multiplication of two 2-dimensional matrices. The benchmark tests multiple calls to the same function, nested function calls and triple-nested loops.
4. `fir` — Finite impulse response filter. Among the tests are an inner loop with varying number of iterations, and loop iteration dependent branching.

San Diego vision benchmark suite

The San Diego vision benchmark suite (SD-VBS) [32], created by the University of California, San Diego, is not directly related to WCET research. It is a benchmark suite intended for the research on the performance and efficiency of computer vision algorithms, in an era of growing numbers of multiprocessor architectures.

The following benchmarks from SD-VBS have been ported to our tool:

1. `disparity` — disparity map computes depth information in an image using dense stereo. It is characterized by the authors as being data intensive.
2. `mser` — maximally stable extremal regions (MSER) is used as a method for detection of similarities between images from different viewpoints. While this benchmark is part of the open source distribution of SD-VBS, it is not described in [32].
3. `svm` — supervised learning method for classification, is a machine learning algorithm. It is characterized by the authors as being computationally intensive.
4. `stitch` — stitch overlapping images using feature based alignment and matching. The benchmark is characterized by the authors as being both data and computationally intensive.

4.3 Overview of technical design

In this section the technical design is presented at the global level. Specific implementation details can be found in Appendix A.

The main components which together form our parametric WCET estimation tool are **(1)** a Raspberry Pi as the experimental platform, **(2)** a computer running the tool's main program and reporting functionality, **(3)** an Arduino

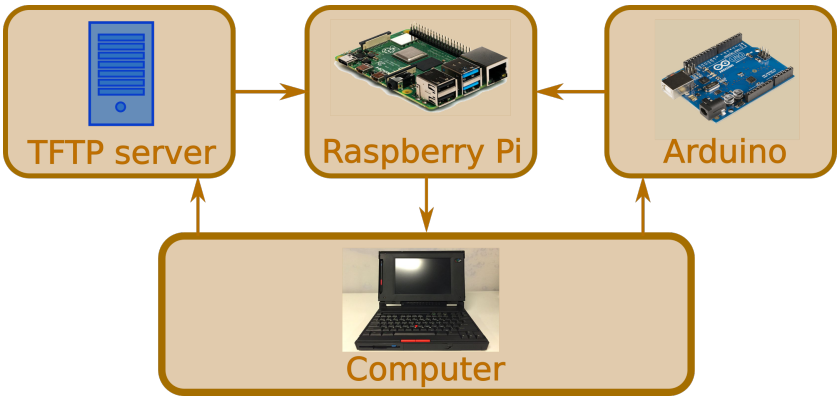


Figure 4.2: **Main components that together form the tool**

that controls the hardware platform's reset pins, and **(4)** a TFTP server which is used to be able to boot the Raspberry Pi from the network. These components are schematically depicted in Figure 4.2.

Raspberry Pi

The heart of our tool is the Raspberry Pi, a popular and economically attractive single board computer [18]. Our tool supports two models of the Raspberry Pi:

- **Raspberry Pi 3 model B** — containing the ARM Cortex A53 quad core processor, featuring in-order execution of instructions;
- **Raspberry Pi 4 model B** — containing the ARM Cortex A72 quad core processor, featuring out-of-order execution of instructions.

Both processors present in the Raspberry Pi 3 and 4 are SoC (system on chip) processor architectures, which implement the ARMv8 instruction set architecture [5].

On startup, the Raspberry Pi starts executing the main scheduling algorithm. This algorithm starts the task set. After each iteration, the results of the execution is written to the computer through a serial connection.

Computer

The computer acts as the control center for the experiments. From the computer the defined experiments, which are administered by the use of a spreadsheet, are read by a Python script that globally takes the following steps, illustrated in pseudo code:

```
foreach line in spreadsheet:
  read parameters from line
  foreach parameter in parameters:
      set compilation option for parameter
  compile sources
  send signal to Arduino to reset the Raspberry Pi
```

```
while input line from Raspberry Pi:
    read input line
    output line to log file
    if enough observations read:
        send signal to Arduino to reset the Raspberry Pi
```

After the experiments have been performed, the user can start the automatic generation of the reports.

Arduino

The Arduino is an easy to use open source electronics platform. It is based on a 8-bit microcontroller, specifically the ATmega328p. The Arduino is added to the platform for the sole purpose of being able to reset the Raspberry Pi by sending it an external signal to the `RUN` header of the Raspberry Pi board.[2]

[2]For detailed instructions, see [25]

Chapter 5

Experimental evaluation

In this chapter, our experimental evaluation is presented. This chapter starts with a description of the experimental setup, in which the evaluation platform and our method for hypothesis testing are explained. Following, three different series of experiments are presented, these are **(1)** the evaluation of the Mälardalen `bsort100` benchmark, **(2)** the evaluation of the SD-VBS `disparity` benchmark, and **(3)** the results of the delayed execution of the Mälardalen `matmult` and the SD-VBS `stitch` benchmarks.

5.1 Experimental setup

As described in Section 4.2.2, our parametric WCET estimation tool consists of three categories of benchmarks. These are synthetic benchmarks, a selection from the Mälardalen benchmark suite and a selection from the San Diego vision benchmark suite.

5.1.1 Choice of benchmarks to evaluate

While our proposed tool contains twelve separate benchmarks in total, we limit the experimental evaluation of the co-runners' shared hardware resource problems to a subset of these benchmarks. The experiments are centered around the research questions and the hypotheses that we want to validate. The hypotheses are:

1. The distribution of the execution time of a task running in isolation is significantly affected by the existence of co-running tasks.

2. The distributions of the execution times of tasks that are concurrently running on multiple cores are significantly affected by the starting time offset of these tasks.

The experimental evaluation is divided in three series of experiments:

- The first hypothesis is investigated by running experiments with the Mälardalen `bsort100` benchmark (Section 5.2.1). The reason for choosing this benchmark is its apparent *insensitivity* to the shared hardware resource contention problem;

- Next, the first hypothesis is shown from another perspective with an extensive evaluation of the SD-VBS `disparity` benchmark (Section 5.2.2). The choice for `disparity` is quite the opposite when compared to the first choice: `disparity` is particularly *sensitive* to the shared hardware resource contention problem;

- Finally, the second hypothesis is evaluated by running experiments with the Mälardalen `matmult` benchmark and the SD-VBS `stitch` benchmark (Section 5.2.3). These experiments are meant to evaluate the second hypothesis, in which delayed execution of co-runners is the central theme.

5.1.2 Experimental platform

As described in Section 4.3, we use two target platforms, the Raspberry Pi 3 model B and the Raspberry Pi 4 model B. To ensure that we have maximum control over the runtime environment, we use a 'bare metal' solution in both cases. We have built on two different open source platforms to run the system, these are `xRTOS` [9] and `circle` [29] for the Raspberry Pi 3 and 4, respectively. The two evaluation platforms are summarized in table Table 5.1.

Bare metal OS	`xRTOS`	`circle`
Single board computer	Raspberry Pi 3 model B	Raspberry Pi 4 model B
Processor	ARM Cortex A53	ARM Cortex A72
Execution	in-order execution	out-of-order execution
RAM	1GB LPDDR2	4GB LPDDR4
L1 cache	32KB	32KB
L2 cache	512KB	512KB

Table 5.1: **Summary of the two experimental platforms**

When a task is run in isolation, we make sure that the other cores are idle. In addition, we make sure that tasks run without preemption, by disabling any interrupts during their execution.

5.1.3 Hypothesis testing

As mentioned in Chapter 1, we evaluate two hypotheses, that are meant to show the existence of the co-runners' effects. These hypotheses are:

1. The distribution of the execution time of a task running in isolation is significantly affected by the existence of co-running tasks.

2. The distributions of the execution times of two tasks that are synchronously running on two cores are significantly affected by the start time offset of these tasks.

Our method for evaluation of the hypotheses is based on standard statistical hypothesis testing, in which a null hypothesis ('H_0') and an alternative hypothesis ('H_a') are considered [13]. The null and alternative hypotheses are competing propositions, they cannot both be true.

Since in general, rejecting a hypothesis is easier than proving a hypothesis, we assume that H_0 is true and try to find evidence for rejecting it. If the statistical test indicates that H_0 can be rejected, we have evidence for suggesting that H_a holds true.

The statistical test's output is a probability called p. This probability stands for the probability of making a *type 1 error*: this error is made when H_0 is incorrectly rejected. By convention, the probability of making a type 1 error is called α, which is set to 0.05. When the result of the test p is lower than α, we say that we have enough evidence to reject H_0.

In our evaluation, we take the two distributions of measured execution times (cycles) that we want to compare with each other. H_0 is defined to be the opposite of the hypotheses that we want to prove, i.e. we assume for H_0 that the distribution of the measured cycles is **not** significantly affected by the presence of co-runners. Since we cannot assume that the data sets are from a normal distribution, we use the non-parametric Mann-Whitney U test for testing H_0 [15].

The first hypothesis is evaluated in the context of the Mälardalen `bsort100` benchmark, the result of which is described in Section 5.2.1. The results of evaluating the second hypothesis is presented in Section 5.2.3.

5.2 Experiments

In this section the evaluation of the hypotheses is presented, together with a study on the factors that influence the WCET of a task that experiences contention for shared hardware resources.

5.2.1 Mälardalen `bsort100`

The Bubblesort implementation by the Mälardalen WCET research group is called `bsort100`. The '100' stands for the number of elements in the original source code. However, this number can be easily extended by a larger amount of elements. The number of elements has a significant effect on the slowdowns due to co-runners. In the following, we present the evaluation of the first hypothesis, for running `bsort100` with 2,000 and with 8,000 elements:

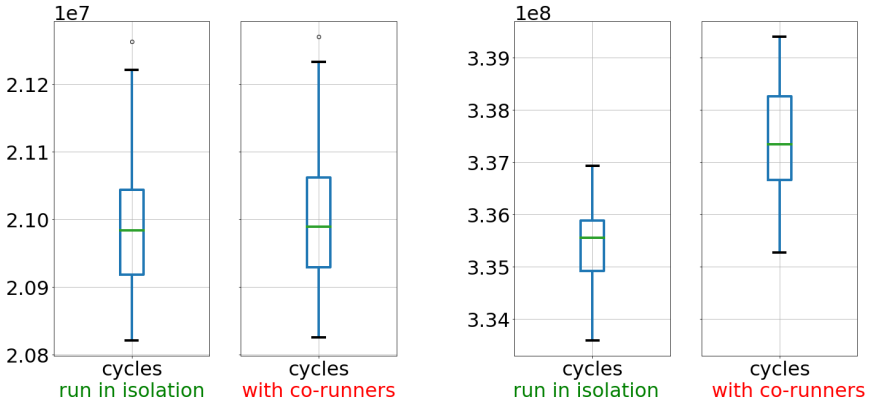
1. The distribution of the execution time of a task running in isolation is significantly affected by the existence of co-running tasks.

`bsort100` — 2,000 elements

With 2,000 elements, the median number of cycles spent by the algorithm when run in isolation is 2.098e7 cycles. The measured WCET amounts to 2.126e7 cycles, which is larger then the median by a factor of 1.013.

When run in parallel with three co-runners that execute the `linear array write` benchmark, the `bsort100` task's measured WCET is **1.00026**× slower than the measured WCET when run in isolation.

In Figure 5.1a, a boxplot is depicted, showing the quartiles of the distribution of measurements. The left boxplot shows the measurements for the task when run in isolation, the right boxplot shows the measurements for the task running with three co-runners.



(a) bsort100 with 2,000 elements **(b)** bsort100 with 8,000 elements

Figure 5.1: **Boxplots of `bsort100` measurement distributions in number of cycles.**

Visually, the boxplots hardly show a difference. When we perform the statistical Mann-Whitney U test on the both data distributions, the calculated p-value is 0.3646. Since this is larger than α, we cannot reject H_0 and we have no evidence for supporting H_a. This means that in the case of `bsort100` running on 2,000 elements, our first hypothesis **cannot** be confirmed.

`bsort100` — 8,000 elements

With 8,000 elements, the median number of cycles spent by the algorithm when run in isolation is 3.36e8 cycles. The measured WCET is 3.37e8 cycles, which is larger then the median by a factor of 1.004.

When run in parallel with three co-runners that execute the `linear array write` benchmark, the `bsort100` task's measured WCET has increased to 3.394e8, which is **1.00734**× slower than the measured WCET when run in isolation.

Figure 5.1b shows the boxplot for the experiment with 8,000 elements. In this case the visual difference is clear between the experiment when run in isolation and with the presence of co-runners. The Mann-Whitney U test now produces a p-value of 4.472e-15, which is very small compared to α. This means that in the case of `bsort100` running with 8,000 elements, H_0 can be rejected and we have evidence to indicate that our first hypothesis holds true.

Evaluating hypothesis 1 with `bsort100`

This evaluation shows that the co-runners' effects are very much dependent on the circumstances. In the case of `bsort100`, the amount of input elements determines whether or not the slowdown can be identified using statistical testing of the measured cycles.

While in the case of 8,000 elements for `bsort100`, statistical methods can identify the experienced slowdown due to co-runners, we have shown using our tool that `bsort100` is rather insensitive to co-runners. For other, real-world tasks a similar evaluation could be performed to gain knowledge on the sensitivity of tasks to co-runners.

5.2.2 SD-VBS `disparity`

The `disparity` benchmark computes depth information in an image using dense stereo. The authors of the benchmark characterize the benchmark as being data intensive. The input data of `disparity` are two 2-dimensional images.

The `disparity` benchmark is an interesting benchmark to include in our study. The experiment with the `disparity` benchmark is chosen for its high level of data intensity. With this benchmark, the co-runners should create a lot of problems for the task running `disparity`. Prior work reported very large slowdowns of factors larger than 300 [6] and 400 [21]. With `disparity`, we expect to be able to find proof for supporting hypothesis 1 under all circumstances.

In this section, the `disparity` benchmark is evaluated on both hardware platforms, starting with the Raspberry Pi 3. On both platforms, `disparity` is being run with the `linear array access` benchmark and with the `linear array write` benchmark.

Running `disparity` on Raspberry Pi 3

Our work attempts to reproduce the slowdown reported by prior work. However, as explained in Chapter 1, our research goal is different. Instead of obtaining the highest possible slowdown, our work attempts to find the factors of influence that steer the co-runners' effects. To this end, we evaluate `disparity` with different input sizes and show that the slowdown factor heavily depends on the size of the input data.

Figure 5.2 shows the results of running `disparity` together with one to three co-runners executing a `linear array access`. The experienced slowdown due to co-runners is 1.9×, the slowdowns seems to be linear in the size of the input data, but a non-linear growth of the slowdown is shown in the number of co-runners.

In the following experiment, far worse slowdowns are measured. Figure 5.3 depicts the results of running the `disparity` benchmark, together with one to

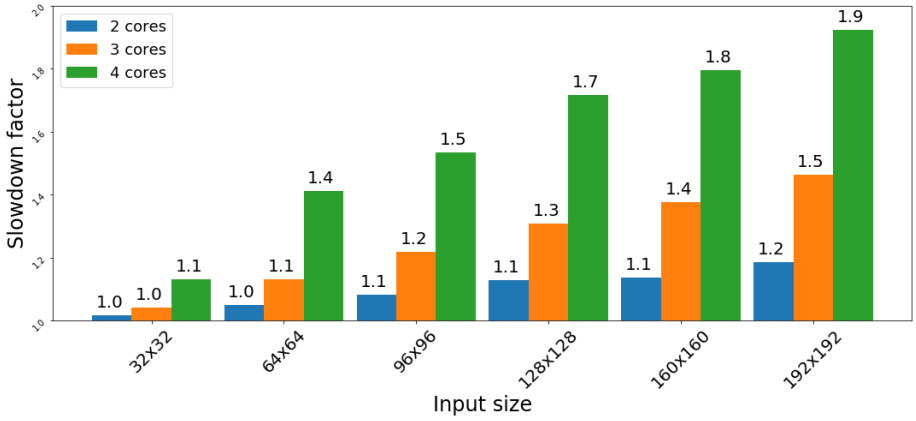


Figure 5.2: **Measured WCET slowdown of `disparity` and `linear array access` run by 1 to 3 co-runners, experiment run on Raspberry Pi 3 model B, with the ARM Cortex-A53.**

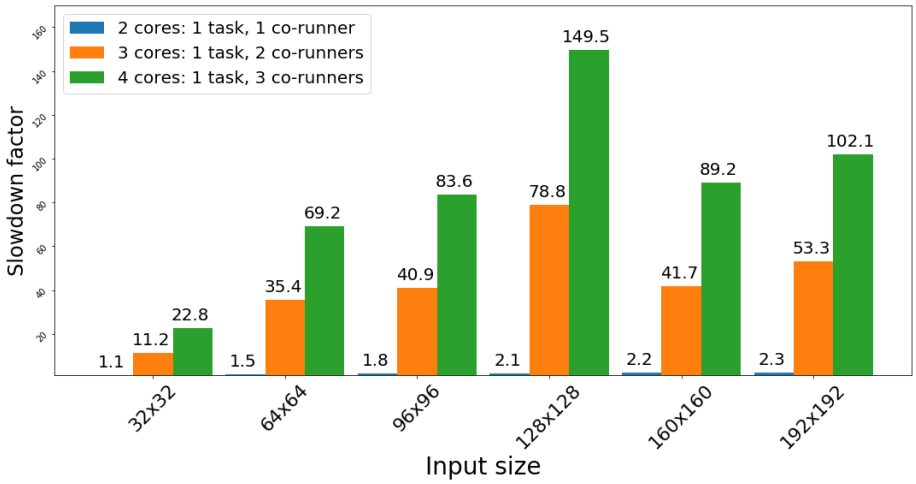


Figure 5.3: **Measured WCET slowdown of `disparity` and `linear array write` run by 1 to 3 co-runners, experiment run on Raspberry Pi 3 model B, with the ARM Cortex-A53.**

three cores running the `linear array write` benchmark. These results show us that:

- The worst slowdown effect that was measured was **149.5**× slower than the same benchmark when run in isolation. This slowdown is much smaller than the slowdown factor reported in [6] and [21]. We suspect that an important factor in creating this difference is the fact that our benchmarks run in a bare metal OS, instead of a full fledged OS such as Linux;

- The figure clearly shows that the worst slowdown occurs with input size 128×128, with three co-runners running in parallel. The input size 128×128 seems to be the sweet spot in terms of co-runners' effects.

- When there is only one co-runner, the slowdown factor is relatively small and linear in the benchmark's input size. For two and three co-runners, the slowdown factor is not linear in its input size anymore.

In an attempt to explain the behavior of the task running in parallel with co-runners, we present some performance metrics in Table 5.2. The PMU event counter is configured to count L2 cache accesses, L2 cache refills, bus accesses and bus cycles. All metrics count the number of times that shared hardware resources are required.

From the performance metrics data, the following observations can be made:

- An important factor in the experienced slowdown is the increased number of L2 cache refills. Clearly, the task experiences shared L2 cache evictions by the co-runners;

- The number of bus accesses are increased for the task when run with co-runners. The task must rely more on slow memory operations;

- The ratio between the number of bus cycles and bus accesses is increased significantly, which shows that memory requests take longer when multiple memory requests are issued in parallel.

Disparity 32x32	cycles 1core	cycles 4cores	slowdown factor
WCET	674,779	15,413,467	**22.842**
L2 cache access	6,349	6,224	**0.980**
L2 cache refill	464	828	**1.784**
Bus access	1,860	5,202	**2.797**
Bus cycles	397,788	7,771,207	**19.536**
Bus cycles/access ratio	213.865	1,493.888	**6.985**
Disparity 128x128	cycles 1core	cycles 4cores	slowdown factor
WCET	10,173,967	1,520,945,739	**149.494**
L2 cache access	534,184	519,889	**0.973**
L2 cache refill	14,204	61,974	**4.363**
Bus access	99,932	434,201	**4.345**
Bus cycles	5,147,525	760,477,260	**147.736**
Bus cycles/access ratio	51.510	1,751.441	**34.002**
Disparity 192x192	cycles 1core	cycles 4cores	slowdown factor
WCET	25,411,431	2,595,104,110	**102.123**
L2 cache access	1,158,374	1,147,208	**0.990**
L2 cache refill	141,800	225,369	**1.589**
Bus access	1,101,063	1,524,813	**1.385**
Bus cycles	12,766,275	1,297,616,331	**101.644**
Bus cycles/access ratio	11.595	851.000	**73.397**

Table 5.2: **Measured cycles and events for `disparity` and 3 co-runners, experiment run on Raspberry Pi 3 model B, with the ARM Cortex-A53.**

Running `disparity` on Raspberry Pi 4

We are also interested in evaluating the `disparity` benchmark on the Cortex-A72 processor. The A72 is a more advanced processor than the A53. It features out-of-order processor cores, which may lead to different results for the evaluation when compared to the A53 with in-order processor cores.

In Figure 5.4, the results of running `disparity` together with `linear array access` co-runners are shown. Interestingly, the slowdown is not linear anymore in the size of the input data. The input size 128×128 forms the worst case, with a measured slowdown of factor 2.4. The slowdowns due to reading co-runners are slightly worse when compared to the same experiment run on the Raspberry Pi 3.

Figure 5.5 shows the result of running `disparity` on the Raspberry Pi 4 with one to three `linear array write` co-runners. These results differ significantly with the same experiment when run on the Raspberry Pi 3. The slowdown effect is far less negative. In an attempt to understand the differences, Table 5.3 lists the same performance metrics as above. The most important difference seems to be the less negative impact of co-runners on the memory bus, when compared to the memory bus of the Cortex A53 processor.

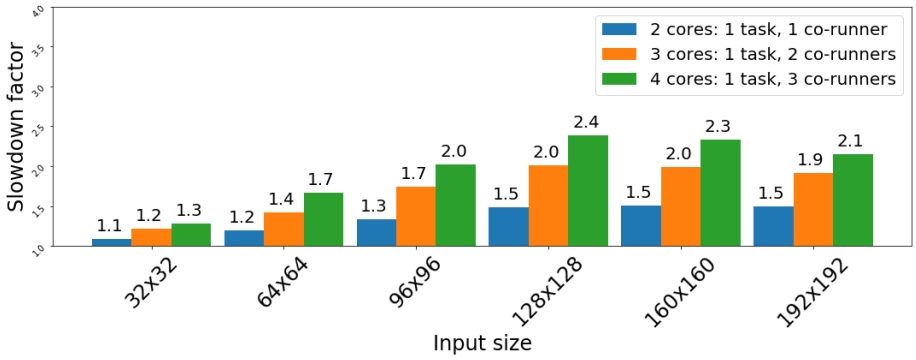


Figure 5.4: **Measured WCET slowdown of `disparity` and `linear array access` run by 1 to 3 co-runners, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

Evaluating hypothesis 1 with `disparity`

For `disparity`, the case for finding evidence to support hypothesis 1 is trivial. In particular, when run on the Raspberry Pi 3, the `linear array write` synthetic benchmark is a co-runner that has a profound effect on the task's WCET.

The evaluation of the same experiments on the Raspberry Pi 4 show a very different result. Evidence for supporting hypothesis 1 is still easily found, but the negative impact of the `linear array write` benchmark is nowhere near the negative impact of the same experiment running on the Raspberry Pi 3.

When looking at the results of the performance events in both cases, the memory bus seems to play an important role. The Raspberry Pi 3 features 1 GB of LP-DDR2 RAM, while the Raspberry Pi 4 features 4 GB of LP-DDR4 RAM. We suspect that the higher throughput of the newer LP-DDR4 memory is much better equipped to deal with many concurrent memory accesses, which could be an explanation for the big differences between the two platforms.

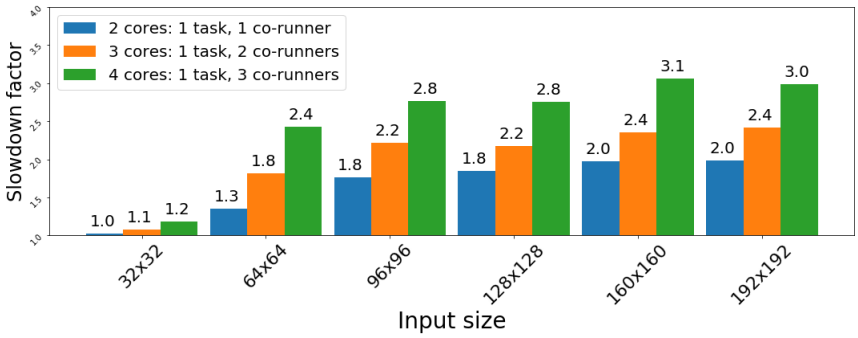


Figure 5.5: **Measured WCET slowdown of `disparity` and `linear array write` run by 1 to 3 co-runners, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

Disparity 32x32	cycles 1core	cycles 4cores	slowdown factor
WCET	465,289	550,539	**1.183**
L2 cache access	14,226	13,703	**0.963**
L2 cache refill	956	827	**0.865**
Bus access	6,380	5,904	**0.925**
Bus cycles	276,806	275,483	**0.995**
Bus cycles/access ratio	43.387	46.660	**1.075**
Disparity 160x160	cycles 1core	cycles 4cores	slowdown factor
WCET	10,317,812	31,610,133	**3.064**
L2 cache access	1,022,754	1,009,905	**0.987**
L2 cache refill	21,524	53,078	**2.466**
Bus access	180,560	484,484	**2.683**
Bus cycles	5,159,270	15,805,088	**3.063**
Bus cycles/access ratio	28.574	32.623	**1.142**
Disparity 192x192	cycles 1core	cycles 4cores	slowdown factor
WCET	18,216,552	54,303,336	**2.981**
L2 cache access	1,394,865	1,289,015	**0.924**
L2 cache refill	87,178	112,496	**1.290**
Bus access	645,796	980,808	**1.519**
Bus cycles	9,108,285	27,152,064	**2.981**
Bus cycles/access ratio	14.104	27.683	**1.963**

Table 5.3: **Measured cycles and events for `disparity` and 3 co-runners, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

5.2.3 Delayed execution

The final set of experiments evaluate the delayed execution of co-running tasks. In this experiment, we assume that the co-runners start after the task under study. This is the central theme of the second hypothesis:

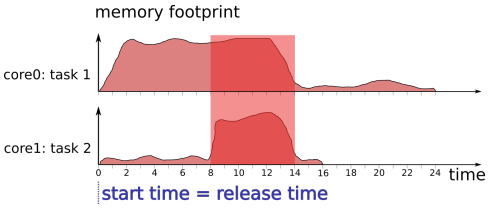
2. The distributions of the execution times of tasks that are concurrently running on multiple cores are significantly affected by the start time offset of these tasks.

In the following, a motivation is presented for the reasoning behind the importance of delayed execution, after which the results of the experiments are presented.

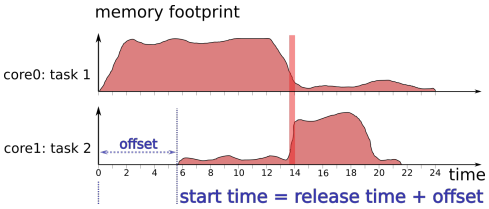
Motivation

The reasoning behind the experiments with delayed execution is an optimization, which in some cases could mitigate the problems created by co-runners. Since the memory hierarchy is largely responsible for the hardware contention, the slowdown effects can be significant when the task has a data intensive profile.

If the data intensity could be visualized graphically as a memory usage profile, we could have the hypothetical situation as depicted in Figure 5.6. The figure shows a potential problem with two cores both running a data intensive task. In Figure 5.6a, the moments in time in which both cores are running a lot of memory operations coincide. However, as tasks could be both memory and computationally intensive, in this hypothetical example task 2 should have a delayed start time as shown in Figure 5.6b. With the delayed start time of task 2, the data intensive sections of both tasks do not coincide, potentially resulting



(a) Without optimization



(b) With optimization

Figure 5.6: **Memory footprints.**

in a lesser slowdown caused by co-runners. The experiments that are described next, show how our parametric WCET estimation tool can be used to analyze tasks running with co-runners, in order to gain information that could be used by a scheduling algorithm.

Evaluation of the second hypothesis

The first experiment within the final set of experiments, is designed to evaluate our second hypothesis, which states that the distribution of execution times of concurrently running tasks are significantly affected by the start time offsets of these tasks. This evaluation is done by running the Mälardalen `matmult` benchmark on the first core with one to three co-runners running both in sync and with varying start time offsets.

The `matmult` task is run on the Raspberry Pi 4. As a *baseline* performance, where the task is run in isolation, the median number of cycles spent on its execution is 4,724,764. When run in sync with three co-runners each running the `linear array write` benchmark, the median of the measured execution time is 7,792,444, which means that `matmult` has become 1.65× slower, when run in sync with three co-runners.

In order to show that the slowdown becomes less when the co-runners have a delayed start time, we conceptually divide the baseline execution WCET in multiple time intervals we call *delay steps*. The number of delay steps determines the *offset*, which is the delay of the start time of the co-runners.

When we divide the baseline WCET by 10, we create delay steps that are equal to $1/10^{th}$ cycles of the baseline WCET. When co-runners have a start time that is equal to the release time plus any number of delay steps, the slowdown effect is expected to become less when compared to all tasks running in sync with each other. Specifically, when the co-runners start their execution after 10 delay steps, `matmult` will have had enough time to finish its execution, without any slowdown.

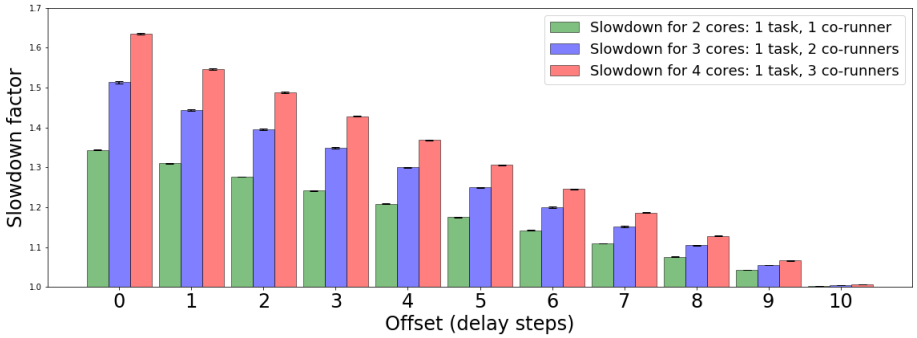


Figure 5.7: **`Matmult` and `linear array write` run by 1 to 3 co-runners, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

The result of this experiment is shown in Figure 5.7. The figure clearly shows a diminishing slowdown with increasing delay of the co-runners' start times. As expected, after 10 delay steps (which is equal to the number of cycles of the baseline), the slowdown is no longer noticeable.

For each offset, 50 iterations are executed. The slowdown depicted is the mean number of cycles for each offset, with a 95% confidence interval around the mean. The confidence levels shown in the figure are very close to the mean, which leads to the conclusion that for the case of the `matmult` task running with `linear array write` tasks as co-runners, the second hypothesis holds true. The distributions of the execution times of `matmult` are significantly affected by the start times of the `linear array write` co-runners.

An interesting feature of the result depicted in Figure 5.7, is the fact that the slowdown seems to show a linear behavior in relation to the number of offsets. This property is described in the following experiment.

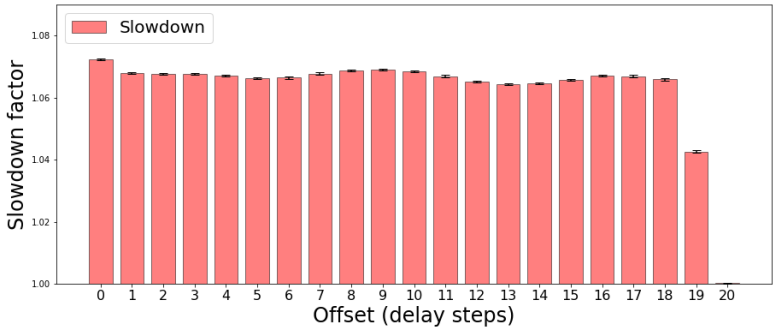


Figure 5.8: **`Matmult` and `linear array write` run with 3 co-runners with increased start times, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

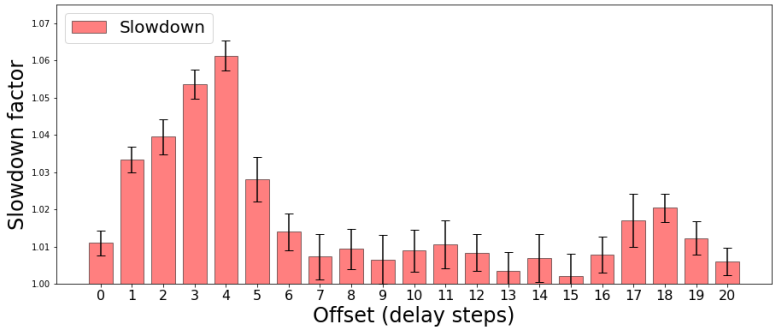


Figure 5.9: **`Stitch` and `linear array write` run with 3 co-runners with increased start times, experiment run on Raspberry Pi 4 model B, with the ARM Cortex-A72.**

Evaluation of a task's data intensity

As described above, one idea that could lead to a potential optimization for a scheduling algorithm, is to make use of knowledge about a task's behavior in terms of the number of memory operations. It seems that data intensive tasks, with `disparity` being a clear example, seem to exhibit a substantial slowdown with co-runners. However, when a task is both data intensive and computationally intensive, the task should show periods of less frequent memory operations during its execution time. These periods could be potentially filled with co-running tasks, where the slowdown effects are mitigated due to the lesser memory operations executing in parallel.

With this experiment, we show that our parametric WCET estimation tool is capable of analyzing the data intensity of tasks. With the goal of obtaining a data intensity profile of a task, we have designed an experiment in which three `linear array write` co-runners are started with varying delayed start times. Instead of executing the co-runners long enough to maximize the worst case slowdown, in this experiment the co-runners are run for a short period of time, specifically for $1/20^{th}$ of the baseline number of cycles. The slowdown experienced by the task under study will depend on the actual data intensity level at the time the co-runners are being run.

Figure 5.7 already gave a hint about the way the data intensity level is distributed in the `matmult` benchmark. In Figure 5.8, the result of the experiment with briefly running co-runners is shown for `matmult`. Indeed, a uniform distribution of the levels of data intensity is shown. As such, the `matmult` benchmark would likely not be an ideal candidate for optimization with delayed execution of co-runners.

However, the `stitch` benchmark from the SD-VBS benchmark suite is described by the authors as being both data and computationally intensive [32]. Therefore, we repeat the same experiment as described above for the `stitch` benchmark, the results of which are shown in Figure 5.9. The `stitch` task seems to show a data intensive behavior at the beginning of its execution, which leads us to believe that in this specific case a delayed execution of co-runners could in fact effectively mitigate the co-runners' problem.

5.3 Summary of the experimental results

In this chapter, we have shown three sets of experiments, in which tasks are run in isolation and with one to three co-runners. The first set of experiments was done with the Mälardalen `bsort100` benchmark. These experiments have shown that `bsort100` is particularly insensitive to the co-runners' problem. This is good news, because it demonstrates that not all programs are susceptible to the co-runners' contention.

The following set of experiments has shown quite the opposite. The SD-VBS `disparity` benchmark appears to be highly sensitive to the co-runners' problem. However, the level of disturbance is dependent on multiple factors. First of all, the hardware architecture seems to play a significant role. Also, the size of the input data is of importance to the severance of the problem.

The final set of experiments is meant as a first step towards a possible mitigation strategy of the shared hardware resource problem. Especially when a task has a mixed profile of both data intensive periods and computationally intensive periods during its execution, a scheduling algorithm could use this knowledge by delaying the execution of one or more co-runners. In this way, a situation in which multiple data intensive tasks are running could be avoided and contention for shared hardware resources could be minimized.

Chapter 6

Conclusions and future work

The WCET estimation problem is a challenging one. Hardware resources that are shared amongst tasks that run in parallel, make the problem even more difficult. While static WCET estimation is an active area of research, it seems that the problem of shared hardware resources in commercial-off-the-shelf (COTS) processors cannot be handled statically yet. Industry makes use of these COTS processors, and therefore it has to rely on measurement-based estimation tools for now.

In this work, we presented a configurable en flexible tool for dynamic measurement of WCETs of tasks. With our parametric WCET estimation tool, system designers are able to create experiments with arbitrary task sets, including multiple co-running tasks. Our tool allows for the configuration of various parameters, which can influence the WCET of tasks.

With our tool, the following research questions were analysed:

- What are the factors that influence the WCET of tasks on a multiprocessor platform?
- How can we quantify these factors of influence?

In an attempt to search for answers to the research questions, we have formulated two hypotheses that we have evaluated by using our proposed tool. These hypotheses were evaluated:

1. The distribution of the execution time of a task running in isolation is significantly affected by the existence of co-running tasks.
2. The distributions of the execution times of tasks that are concurrently running on multiple cores are significantly affected by the starting time offset of these tasks.

Our parametric WCET estimation tool consists of three sets of selected benchmarks, that can be evaluated on two alternative hardware platforms. These hardware platforms were chosen for their specific processor architectures. They contain an ARM Cortex A53 featuring an in-order quad core processor and an ARM Cortex A72 featuring an out-of-order quad core processor. The operating

systems on the hardware platforms are 'bare metal' systems, as such they can be manipulated relatively easy, but at the same time can be harder to debug.

The goal of the tool is to be able to easily run a multitude of experiments, which can be tuned by the use of parameters. Examples of parameters that are an important factor in the resulting execution times of tasks, are the set of co-running tasks that run in parallel and the sizes of the input data that the tasks process. Our tool also incorporates more specialized parameters, such as whether or not to enable the memory management unit (MMU) and the specific performance metrics to capture.

The three chosen benchmark suites that are included in our tool are **(1)** synthetic benchmarks which are designed to stress the memory system, **(2)** a subset of the Mälardalen WCET benchmarks and **(3)** a subset of the San Diego vision benchmark suite (SD-VBS). With these benchmarks, a number of experiments have been designed with which the research questions and hypotheses were evaluated.

Evaluation of the first hypothesis

To evaluate the first hypothesis, the Mälardalen `bsort100` (bubblesort) algorithm was chosen to run in parallel with three co-running tasks. With the first hypothesis we wanted to show that the distribution of execution times of tasks is significantly affected by the presence of co-running tasks. The `bsort100` benchmark is an interesting choice for the evaluation of the first hypothesis, because of its insensitivity to co-runners. For `bsort100`, in some cases the use of statistical testing methods is needed prove or disprove the contention for shared hardware resources.

The experiment has been performed with two different data input sizes. For the case where `bsort100` was executed with an input size of 2,000 elements, the resulting execution times did not show a significant difference when compared with execution times of the task when run in isolation. In other words, we do not have enough evidence to support the first hypothesis for the `bsort100` benchmark with input size of 2,000 elements.

However, when we increased the input data size to 8,000 elements, the statistical hypothesis test showed a significant result. Although hardly noticeable when looking at the measured slowdown of the experiment with co-runners (1.00734×), the Mann-Whitney U test produced a p-value of 4.472e-15. With α set to 0.05, we conclude that for the case of running `bsort100` together with three co-runners and an input data size of 8,000 elements, the first hypothesis holds true.

The first set of experiments with the `bsort100` benchmark showed us that the input data size is an important factor in the problem of shared hardware resource contention. This result has been further analysed in detail, by evaluating a different benchmark. The `disparity` benchmark from the SD-VBS benchmark suite is central in our second set of experiments.

More extensive evaluation of data input sizes

The `disparity` benchmark is characterized as being data intensive by its creators, and prior studies show that `disparity` experiences serious slowdowns because of the shared hardware resource contention. It was therefore chosen as

the subject for our second set of experiments, in which we evaluated the effects of co-runners with varying input data sizes.

The `disparity` benchmark was run together with one to three co-running tasks, each running either the `linear array access` benchmark or the `linear array write` benchmark. The experiments have been done on both hardware platforms. The resulting measured execution times were greatly affected by the presence of co-runners, especially in the case of `disparity` running together with three `linear array write` tasks. The worst slowdown of 149.5× was measured with an input size of 128×128. Both smaller and larger input data sizes exhibited less slowdowns.

Interestingly, the Raspberry Pi 4 with the out-of-order processor cores showed far less pessimistic results. Here, the worst case execution time measured for `disparity` was slower by a factor of 3.06, when compared to the same task when run in isolation. We suspect that the improved memory type (LP-DDR4) is much better equipped to handle simultaneous memory requests, when compared to the older memory type present in the Raspberry Pi 3 (LP-DDR2).

The results of the measured slowdowns which are non-linear in the size of the input data, seem to suggest that the shared L2 cache is used quite effectively for some sizes of input data in the case of a single task running in isolation. When the L2 cache is stressed by multiple co-runners, the effective use of the L2 cache is gone and the measured WCET is increased with the presence of co-runners.

Delayed execution of co-runners

In the experiments described above, the execution of co-runners was synchronized with the task under study. In other words, the start times of the task and its co-runners were equal. With equal start times, we expected to maximize potential co-runners' effects, because the co-runners were run for as long as the task under study ran.

In order to evaluate the second hypothesis, a final set of experiments was created to delay the execution of co-runners. Our second hypothesis states that the distribution of execution times of a task is significantly affected by the start times of the co-running tasks.

The knowledge about potential slowdowns as a function of the co-runners' start times could be of importance to a scheduling algorithm, that could delay the execution the co-runners as an optimization strategy. We have shown that our parametric WCET estimation tool can be used to investigate tasks in terms of their sensitivity to delayed co-runners. Specifically, we have shown that for a task that exhibits both data intensive and computationally intensive behavior, our tool is able to show this behavior in a 'memory usage' profile. This profile is created by targeting the task under study at specific time intervals, where at each time interval the co-runners attempt to stress the task for a short period of time.

In these experiments we selected the `matmult` and the `stitch` benchmarks. From the results we conclude that `matmult` shows a uniform slowdown behavior during its execution. This suggests that the level of data intensity does not vary much. Hence, for `matmult`, an optimization as proposed would likely not be beneficial.

However, the `stitch` benchmark exhibits a mixed data and computationally intensive behavior. Indeed, for `stitch` our measurements showed that the data

intensity level fluctuates during its execution. At the beginning of its execution, `stitch` shows data intensive behavior. This leads to the belief that for `stitch`, a scheduling algorithm may be able to mitigate any contention for shared hardware resources, by delaying the execution of co-runners.

Parametric WCET estimation tool

For this work, we have created a parametric WCET estimation tool and have performed a number of experiments with it. Our tool is open source and publicly available. This includes the data and data processing scripts that were created for the reported experiments. Our aim for the tool is to be able to create both configurable and reproducible experiments.

Any experienced computer user or computer programmer should be able to use our tool. Users should be aware of the fact that two operating system platforms have been chosen to port the benchmarks to. This was a pragmatic choice. It was necessary to combine the two platforms in one tool to be able to cover the two Raspberry Pi versions. As described below, a future version of the tool should feature an integrated solution for the underlying operating system. Appendix A describes the inner workings of the tool in detail, including some known limitations.

6.1 Future work

The following improvements to our parametric WCET estimation tool are left for future work. These improvements fall in three categories. First of all, the features of the two operating systems on which our tool is based should be integrated. Ideally, only one operating system is used that supports different computer architectures. This would lead to a simpler tool and more opportunities to compare results of different processors to each other.

The second category of improvements concerns the extension of parameters that can influence the effects of contention for shared hardware resources. Examples of such parameters are the use of virtual memory, compiler optimization settings and configuration of hardware prefetchers.

Finally, we note that our tool currently only works with random data as input data. For a more reliable estimation of WCETs, the user of the tool should have a finer control over input data sets. Instead of randomly generating input data, the platform should be able to receive input data values prior to starting the execution of the tasks.

Bibliography

[1] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. A survey of industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[2] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis*, Lecture Notes in Computer Science, pages 52–66, 1996.

[3] Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, and Christine Rochange. WCET and mixed-criticality: What does confidence in WCET estimations depend upon? In *Proceedings of the 15th International Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 65–74, 2015.

[4] Björn Andersson, Hyoseung Kim, Dionisio De Niz, Mark Klein, Ragunathan (Raj) Rajkumar, and John Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Transactions on Embedded Computing Systems*, 17(3):1–29, 2018.

[5] ARM. ARM architecture reference manual ARMv8, for ARMv8-a architecture profile, 2019. URL `https://developer.arm.com/documentation/ddi0487/fc/`. Retrieved September 15th 2020.

[6] Michael G. Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS), Montreal, QC, Canada, April 16-18, 2019*, pages 357–367. IEEE, 2019.

[7] M.D. Bennett and Neil C. Audsley. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 183–190, 2001.

[8] Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in WCET analysis. In *Proceedings 29th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76, pages 26:1–26:22, 2017.

[9] Leon de Boer. LdB-ECM/raspberry-pi-multicore. URL `https://github.com/LdB-ECM/Raspberry-Pi-Multicore`. Retrieved September 15th 2020.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, 2012.

[12] Robert I. Davis, Sebastian. Altmeyer, Leandro S. Inusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-time systems*, 54(3):607–661, 2018.

[13] Michel Dekking, Cor Kraaikamp, Rik Lopuhaä, and Ludolf Meester. *A modern introduction to probability and statistics: understanding why and how*. Springer texts in statistics. 2005.

[14] Jean-françois Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[15] Michael P. Fay and Michael A. Proschan. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics Surveys*, 4:1–39, 2010.

[16] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999.

[17] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quinones, Tullio Vardanega, and Francisco J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Transactions on Computers*, 66(4):586–600, 2017.

[18] Raspberry Pi Foundation. Teach, learn, and make with raspberry pi. URL `https://www.raspberrypi.org/`. Retrieved September 15th 2020.

[19] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010*, volume 15 of *OASICS*, pages 136–146, 2010.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Fourth edition, 2007.

[21] Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and steady: Measuring and tuning multicore interference. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 200–212, 2020.

[22] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 195–201, 1998.

[23] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):1–48, 2016.

[24] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys*, 52(3):1–38, 2019.

[25] max (blog author). Waking up raspberry pi using reset pin. URL `https://scribles.net/waking-up-raspberry-pi-using-reset-pin/`. Retrieved September 15, 2020.

[26] John L. Patterson, David A. Hennessy. *Computer Organization and Design.* Fourth edition, 2012.

[27] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization*, 8 (4):1–25, 2012.

[28] Eric Sprangle and Douglas M. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 25–34, 2002.

[29] Rene Stange. rsta2/circle. URL `https://github.com/rsta2/circle`. Retrieved September 15th 2020.

[30] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[31] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

[32] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009.

[33] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation*, Communications in Computer and Information Science, pages 430–444, 2008.

[34] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3), 2008.

[35] Meng Xu, Linh T. X. Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356, 2019.

[36] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

Appendix A

Implementation details

This chapter describes the details of the implemented tool. Our tool is open source and publicly available on GitHub[1]. With the description below, users should be able to use the tool and extend it for their own research purposes.

A.1 Technical overview

Globally speaking, our parametric WCET estimation tool consists of four components. These are:

- **Raspberry Pi** — This is the hardware on which the experiments are performed.
- **Computer** — The computer acts like the control center, from which the experiments are run.
- **Arduino** — The Arduino serves as an extension to the computer, with which the Raspberry Pi can be reset.
- **TFTP server** — The TFTP server contains the runtime binary that is downloaded by the Raspberry Pi through the network.

These components are graphically illustrated in Figure A.1 and are further described below.

A.1.1 Raspberry Pi

In the project, two versions of the Raspberry Pi computer form the heart of the experiments. They were chosen for the quad processors that they contain, the ARM Cortex A53 (Raspberry Pi 3) and the ARM Cortex A72 (Raspberry Pi 4).

The connections from and to the Raspberry Pi are:

- Raspberry Pi $\longleftrightarrow$ TFTP server — This is a network connection made with a UTP cable.

[1] https://github.com/cassebas/run-co-runners.git

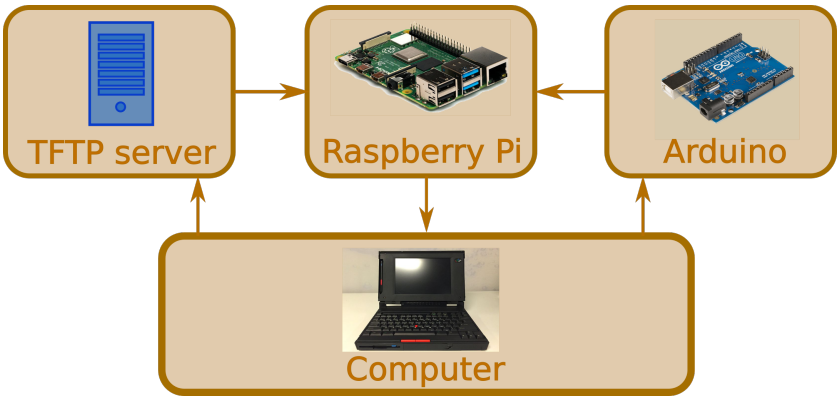


Figure A.1: **Main components that together form the tool**

- Raspberry Pi ⟷ Arduino — This is a 2-wire connection made from the Raspberry Pi's `RUN` header to the Arduino's pin 13 (high/low voltage) and the Arduino's GND (low voltage).
- Raspberry Pi ⟷ Computer — This is a serial connection, made with a USB to TTL serial cable. On the Raspberry Pi, the cable is connected to the `UART0_TXD` and `UART0_RXD` with `GPIO14` and `GPIO15`, respectively. The black `GND` wire in the cable is connected to `GND`, but the red power wire is not connected, since the Raspberry Pi is powered by the official power adapter.

A.1.2 Arduino

The Arduino was added to the experimental platform, to be used as a proxy on behalf of the computer's control center. The Arduino runs a simple program. It will continuously set pin 13 to high and listen to the serial port to receive a character. When the character 'r' is received, it will set pin 13 to low. This will be cause the connected Raspberry Pi's RUN pins to become low, effectively making the Raspberry Pi perform a system reset.

A.1.3 TFTP server

The Raspberry Pi features several options for loading the runtime binary upon power up. The most obvious and easiest alternative is to boot from a microSD card. This method is very simple, it only involves copying the boot files onto a microSD card, put it into the microSD card slot and power on the Raspberry Pi.

Obviously, when doing many experiments, this method quickly becomes infeasible because for each experiment a newly compiled runtime binary is to be copied onto the microSD card. Another solution is to program the Raspberry Pi with a JTAG capable programmer. While the JTAG programmer could be used halt the processor and start a debug session, the method is complex and too involved for just booting the Raspberry Pi.

Finally, there is the possibility for the Raspberry Pi to boot from the network. This method is straightforward and has been used in this project. For this

method to work, a TFTP server is necessary which serves the necessary boot files. The TFTP server could be hosted in the local LAN on a physical device (possibly another Raspberry Pi), but in this project a virtualization platform is used. A virtual machine was created by the use of `vagrant` with `virtualbox` as the provider of the virtual machine.

Details of the implemented TFTP server solution can be found on GitHub[2].

A.1.4 Computer

The computer acts as the control center, on which all experiments are created and run. The experimental platform that runs on the computer is discussed in more detail below.

The computer's connections are:

- Computer $\longleftrightarrow$ TFTP server — Depending on the location of the TFTP server, this could either be a network connection, or it could be just local file copy in the case of a virtualized TFTP server that runs on the same computer.
- Computer $\longleftrightarrow$ Arduino — This connection is a serial connection, which is made with the standard blue Arduino USB cable.

A.2 Implementation of system and benchmarks

As described in Chapter 4, two target platforms are used for running the experiments. Both platforms feature a Raspberry Pi computer, a 'bare metal'-like operating system and the benchmarks that were ported to the platforms. The platforms are described below.

A.2.1 Raspberry Pi 3 + `xRTOS`

The `xRTOS`[3] operating system was especially written for the multicore Raspberry Pi to serve as a basic real-time operating system. The Raspberry Pi versions 2 and 3 are supported. Its implementation is a combination of assembly and C. It features a preemptive scheduler, which in its basic implementation only draws progress bars onto the screen. For the purposes of this project, the preemptive scheduler was modified to allow for non-preemptive execution of the benchmarks. The `xRTOS` repository was forked to apply our experimental additions.[4]

A.2.2 Raspberry Pi 4 + `circle`

The `circle`[5] platform is a bare metal programming environment for the Raspberry Pi. It was written to serve as a educational tool, which can be tried and tested and extended. It supports the Raspberry Pi versions 2, 3, 4 and Zero. It is written in a combination of assembly and C++. `Circle` features a lot of demo

[2] https://github.com/cassebas/Raspberry-Pi-Networkboot.git
[3] https://github.com/LdB-ECM/Raspberry-Pi-Multicore.git
[4] https://github.com/cassebas/Raspberry-Pi-Multicore.git — `experiment` branch
[5] https://github.com/rsta2/circle.git

programs. Also, it contains a structure in which the user's own program can be added with ease. Multi-threaded execution on multiple cores is supported. The `circle` repository was forked to apply our own experimental additions.[6]

A.2.3 The `benchmark_config.m4` script

Both `xRTOS` and `circle` have been extended/modified to continuously run selected benchmarks, while measuring the execution cycles from start to end. The porting and addition of the benchmarks' source code has been done with the requirements of extensibility and maintainability in mind. For this, the use of `m4` macros was chosen.

With the use of `m4` macros, the experiments are configurable can be extended with new benchmarks or programs relatively easy. The main source code does not need be changed for configuration or addition of benchmarks. The way this works is by generation of source code with the use of the `benchmark_config.m4` script. Examples of important macros that are generated are:

- `BENCH_INIT1_CORE0` — This macro will generate code that typically declares or defines variables, which in this case run on core 0. The `INIT1` part in the name stands for the fact that the generated content by this macro is executed once, before the main control loop of the core running the benchmark.
- `BENCH_INIT2_CORE0` — This macro will typically initialize variables, on core 0 in this case. The `INIT2` part in the name stands for the fact that the generated content is executed as part of the main control loop of the core running the benchmark.
- `DO_BENCH_CORE0` — This macro will start the actual execution of the benchmark, that is configured to run in core 0. It is executed on each iteration of the main control loop, thereby generating repeated executions of the same benchmark.

An example usage of the `benchmark_config.m4` script is:

```
$ m4 -Dconfig_series=3111 -Dconfig_benchmarks=1444 -Dinputsize_core0=32 \
-Dinputsize_core1=1024 -Dinputsize_core2=1024 -Dinputsize_core3=1024 \
-Dexp_label=DISPARITY_4CORE_TEST benchmark_config.m4 > benchmark_config.h
```

The result of the `m4` command above is written to the `benchmark_config.h` file. This C header file contains the configution of the benchmarks, like the specific benchmarks to run on which core, memory sizes and so forth. To include the `benchmark_config.h` header file, the environment variable `BENCHMARK_CONFIG` must be set to `-DBENCHMARK_CONFIG_M4` in the make process. This way the C preprocessor knows that it must include the header file.

Below the `benchmark_config.m4` script parameters are described.

Parameters of `benchmark_config.m4`

The `benchmark_config.m4` can take the following parameters:

[6] https://github.com/cassebas/circle.git — `experiment` branch

- `exp_label` — Set the label for the experiment, which will be used in the reported log lines. The label is of importance in the data processing step, which is described in Appendix A.4.

- `config_series` — In the reference implementation of the tool, three types of benchmark series have been implemented. These are **(1)** synthetic benchmarks, **(2)** benchmarks from the Mälardalen benchmark suite and **(3)** the SD-VBS benchmark suite. The selection of the benchmarks series to run is encoded in the `config_series` string, where the length of the string specifies the number of cores that is used in the experiment and the i^{th} digit in the string specifies the series number that is to be run on core i (where $0 <= i <= 3$).

- `config_benchmarks` — The `config_benchmarks` string encodes the benchmark to run. Like in the `config_series` string, its length specifies the number of cores to run. The i^{th} digit in the string specifies the benchmark to be run on core i (where $0 <= i <= 3$). See Table A.1 for the list of implemented benchmarks and their corresponding numbers.

- `inputsize_core`$_n$ — This parameter specifies the input size of the benchmark that is to run on core number n ($0 <= n <= 3$).

- `pmu_core`$_n$ — The `pmu_core`$_n$ specifies the event types that are to be monitored by the ARM performance monitor on core number n. The `pmu_core`$_n$ parameter is a string, where the length of the string is equal to the number of events that must be monitored. The string is encoded by the use of a mapping from an event code (0 to 9) to the event number specified by ARM. The i^{th} position of the string specifies the i^{th} event number to be monitored. See table Table A.2 for the supported event types that can by monitored by the PMU. Currently, the maximum number of events that can be encoded in the string is 4.

- `mmu_enable` — [`xRTOS` only] This parameter configures the MMU (memory management unit) to be enabled. Without specifying this parameter, the MMU will not be enabled (on the `xRTOS` platform).

- `screen_enable` — [`xRTOS` only] This parameter configures the screen to be enabled. Without specifying this parameter, the screen will not be used (on the `xRTOS` platform).

- `delay_step_countdown` — The `countdown` function is a simple function written in assembly that is designed to make the processor spinlock for a specific number of processor cycles. In order to create (reasonably) precise

	1	2	3
1	linear array access	mälardalen bsort100	sd-vbs disparity
2	linear array write	mälardalen ns	sd-vbs mser
3	random array access	mälardalen matmult	sd-vbs svm
4	random array write	mälardalen fir	sd-vbs stitch

Table A.1: **Overview of ported benchmarks**

delays in the start time of the co-runners' execution, the `countdown` function is used in combination with the `delay_step_countdown` parameter. This is further explained in Appendix A.3.

- `report_cycles_countdown` —Since the number of cycles that is spent on each countdown step depends on the specific processor and operating system, the `report_cycles_countdown` parameter can be used to test the `countdown` function and print the number of cycles that is used for the `countdown` function.

- `synbench_repeat` — This parameter allows the co-runners to have an extended duration of execution time, to be able to keep stressing the task under study when the task is running for a very long time. When not specified, `synbench_repeat` is defined as 1.

- `debug_enable` — This parameter enabled debug information to be printed to the serial port.

A.3 Experimental setup

In this section, the main components of the experimental setup are described. First, there is the definition of the experiments by the use of a spreadsheet. Second, the `run_experiments.py` Python script reads the experiments definition and automatically runs multiple experiments and logs all incoming data to an output file. These components are discussed next.

A.3.1 Experiments definition

In this section, the way the experiments are defined is explained. First of all, an important concept for the delayed execution of co-runners is explained.

Event code	ARM event number	Event name
0	0x03	L1 Data cache refill
1	0x04	L1 Data cache access
2	0x05	L1 Data TLB refill
3	0x13	Data memory access
4	0x15	L1 Data cache Write-back
5	0x16	L2 Data cache access
6	0x17	L2 Data cache refill
7	0x18	L2 Data cache Write-back
8	0x19	Bus access
9	0x1D	Bus cycles

Table A.2: **Supported PMU events that can be monitored**

Delayed execution of co-runners

The parametric WCET estimation tool runs the task and its co-runners both in synchronized fashion and with delayed execution of the co-runners. The idea behind the delayed execution is as follows. Because we want to be able to control the length of the delay, the baseline WCET is conceptually divided in time intervals. The baseline WCET is the maximum measured number of cycles that the task needs to complete its task when run in isolation.

The time intervals are called *delay steps*. When the baseline WCET is conceptually divided into 10 delay steps, a delayed execution of 10 delay steps effectively means that the task and co-runners are executed consecutively, instead of in parallel.

The controlling of the delayed execution is done by the parameters `delay step countdown`, `measured wcet baseline` and `cycles per count` parameters in the experiment definition. These parameters and all other parameters are described below.

Explanation of the spreadsheet

For defining the experiments, an Excel spreadsheet is used. On each row, one experiment can be defined, where the columns contain the parameters that define the specifics of the experiment.

The columns with the parameter definitions are:

- `experiment number` — The number of this experiment, this number serves as an identifier for the experiment and is used for selection by the `run_experiments.py` script.
- `platform` — The platform on which the experiment is to be run. This can by either `xRTOS` or `circle`. Please note that currently, all experiments defined in one spreadsheet must run on the same platform.
- `raspberry pi` — The Raspberry Pi version to run the experiment on. This can either be 3 or 4, for running on the Raspberry Pi 3 or Raspberry Pi 4, respectively. Please note that currently, all experiments defined in one spreadsheet must run on the same Raspberry Pi.
- `benchmark_series` — Currently, three types of benchmark series have been implemented. These are **(1)** synthetic benchmarks, **(2)** benchmarks from the Mälardalen benchmark suite and **(3)** the SD-VBS benchmark suite. The selection of the benchmarks series to run is encoded here, where the length of the string specifies the number of cores that is used in the experiment and the i^{th} digit in the string specifies the series number that is to be run on core i (where $0 <= i <= 3$).

  See Table A.1 for an overview of the benchmark series that can be used. Please note that the string is surrounded by quotes (**not** smart quotes), to force a string data type in Excel.
- `benchmark_configuration` — The `benchmark_configuration` string encodes the benchmarks to run. Like in the `benchmark_series` parameter, its length specifies the number of cores to run. The i^{th} digit in the string specifies the benchmark to be run on core i (where $0 <= i <= 3$).

See Table A.1 for the list of implemented benchmarks and their corresponding numbers. Please note that the string is surrounded by quotes (**not** smart quotes), to force a string data type in Excel.

- `enable mmu` — Wheter or not to enable the MMU (memory management unit). This parameter must be `TRUE` or `FALSE`. Please note that this parameter is only supported for the `xRTOS` platform running on the Raspberry Pi 3.
- `enable screen` — Whether or not the enable the screen. This parameter can either be `TRUE` or `FALSE`. Please note that this parameter is only supported for the `xRTOS` platform running on the Raspberry Pi 3.
- `no cache management` — Each experiment is repeated for a multiple iterations (the minimum of which can be specified on the command line when using the `run_experiments.py` script). The default behavior is to clean the instruction and data caches before running each benchmark, in an attempt to create equal conditions between each iteration. The `no cache management` parameter can be used to disable the cache cleaning.
- `experiment label` — Define the experiment label for the experiment, which is used in the data processing step (further described in Appendix A.4).
- `pmu core`$_n$ — The `pmu core`$_n$ parameter specifies the event types that are to be monitored by the ARM performance monitor on core number n. The `pmu core`$_n$ parameter is a string, where the length of the string is equal to the number of events that must be monitored. The string is encoded by the use of a mapping from an event code (0 to 9) to the event number specified by ARM. The i^{th} position of the string specifies the i^{th} event number to be monitored. See table Table A.2 for the supported event types that can by monitored by the PMU. Currently, the maximum number of events that can be encoded in the string is 4.
- `inputsize core`$_n$ — This parameter specifies the input size of the benchmark that is to run on core number n ($0 <= n <= 3$).
- `delay step countdown` — The `countdown` function is a simple function written in assembly that is designed to make the processor spinlock for a specific number of processor cycles. In order to create (reasonably) precise delays in the start time of the co-runners' execution, the `countdown` function is used in combination with the `delay step countdown` parameter.

  This parameter specifies the number of times the `countdown` function must be called, to create one delay step. Please note that this parameter is auto filled in by a formula.
- `measured wcet baseline` — To determine the number of times the `countdown` function must be called for one delay step, the length of the baseline WCET in cycles must be specified. This implies that to correctly create delay offsets for the co-runners, the specification of the experiments is like a 2-stage rocket. First the benchmarks must be run in isolation, to determine the estimation of the baseline WCET (cycles). This number must then be put into the experiment definitions, to be able to compute the number of calls to the `countdown` function to create one delay step.

- `cycles per count` — The number of cycles that is spent for one execution of the `countdown` function. Since the number of cycles that is spent on each countdown call depends on the specific processor and operating system, the `report_cycles_countdown` parameter of the `benchmark_config.m4` script can be used to test the `countdown` function and print the number of cycles that is used for the `countdown` function.

- `synbench_repeat` — This parameter allows the co-runners to have an extended duration of execution time, to be able to keep stressing the task under study when the task is running for a very long time. When not specified, `synbench_repeat` is defined as 1.

An important concept of the Excel spreadsheet is that one spreadsheet should contain both the experiment with the task run in isolation, as well as the experiment(s) with the same task running with one to three co-runners. This way, the `slowdown_factors.py` script (Appendix A.4.2) is able to match each co-runners experiment to its task-in-isolation counterpart.

A.3.2 The run_experiments.py script

The `run_experiments.py` Python script is used to automatically run multiple experiments in one go. The script reads the experiments definition from a spreadsheet, and writes the received log output to a specified output file.

An example usage of the `run_experiments.py` script is:

```
$ python run_experiments.py --working-directory-circle=../../circle/app/corunners \
  --input-file xlsx/experiments_SD-VBS_stitch_circle_pi4.xlsx \
  --output-file output/experiments_SD-VBS_stitch_circle_pi4_exp11.log \
  --min-observations 200 --experiment-begin 11 --experiment-count 1
```

The parameters of the of the `run_experiments.py` script are:

- `--input-file` — The path and name of the Excel input file containing the experiment definitions.

- `--output-file` — The path and name of the output file, to which all logs must be written.

- `--working-directory-xrtos` — The path of the directory where the `xRTOS` system is located. The `xRTOS` system is a submodule of the `run-co-runners` Git repository, by default this parameter is set to `../platforms/raspberrypi/Raspberry-Pi-Multicore/xRTOS_MMU_SEMAPHORE`

- `--working-directory-circle` — The path of the directory where the `circle` system is located. The `circle` system is a submodule of the `run-co-runners` Git repository, by default this parameter is set to `../platforms/raspberrypi/circle/app/corunners`

- `--tty-reset` — Serial port to which the Arduino is connected. It is the path and name of the serial port that is used for the resetting the Raspberry Pi. By default, it is set to `/dev/ttyUSB0`.

- `--tty-logging` — Serial port to which the Raspberry Pi is connected. It is the path and name of the serial port that is used to receive all logging information sent by the Raspberry Pi. By default, it is set to `/dev/ttyUSB1`.
- `--min-observations` — Minimum number of observations that must be seen in the logs received from the Raspberry Pi, before the next experiment can be selected and the Raspberry Pi can receive a reset signal.
- `--experiment-begin` — The experiment identifier of the experiment with which to begin running the experiments.
- `--experiment-count` — The number of experiments that must be run consecutively, starting from the first experiment specified by identifier `--experiment-begin`.
- `-v, --verbosity` — Define the verbosity for the program, which can be either CRITICAL, ERROR, WARNING, INFO or DEBUG. By default, the verbosity level is set to INFO.
- `--help` — Print usage information and exit program.

A.4 Data processing

In this section the data processing step is discussed. The experiments output log data to the serial port, which is captured by the `run_experiments.py` script. This output is converted to several other output formats, such as CSV files and graphical output of measured cycles and PMU events.

A.4.1 Overview of the data processing step

A major part of the data processing is done automatically, by the use of a `Makefile` script that executes the scripts one by one. These scripts are explained in detail below. Here, an overview of the data conversion is described.

The output of the experiments are plain text log files. Each line contains one measurement of the experiment, this can either be a cycles measurement or one of the PMU performance event counter measurements. The plain text log files are first converted to CSV format. They are split into cycles data CSV files and events data CSV files. By default, the log files and CSV files are located in the `run-co-runners/experiment/output` directory.

The above log files and CSV files contain multiple experiments' data in a single file. These files are further separated into files containing the data of only one experiment. Two types of CSV file are created, one containing all data measurements (cycles or events) for one experiment, and the other containing aggregated summary information of the measurements for one experiment (cycles only). The aggregated values are the median, mean and maximum values, including their standard deviations. By default, the location of the CSV files with single experiments is the `run-co-runners/experiment/report/data` directory.

Next, for each experiment a data visualization is generated, using as input the data CSV files containing the cycles measurements. When PMU events

data for the same experiment is present in the same directory, these data will be included in the data visualization.

A separate step is the generation of a PDF report containing graphical output of the aggregated cycles data information. The generation is done using LaTeX templates, which are in itself generated by `m4` macros. The figures in the PDF are generated using the PGFPlots package.[7]

A.4.2 Description of data processing scripts

In the following, the scripts that convert the log data are explained. Most scripts are included in a `Makefile` for automatic processing, except for the `slowdown_factors.py` script.

log2csv-cyclecount.awk

The output log file contains the raw data, where both cycles data and performance events are present. The `log2csv-cyclecount.awk` script captures the cycles data and converts the data to CSV format.
An example execution of the `log2csv-cyclecount.awk` is:

```
awk -f log2csv-cyclecount.awk \
output/experiments_SD-VBS_stitch_circle_pi4-exp11.log
```

log2csv-eventcount.awk

The `log2csv-eventcount.awk` script captures the performance events data and converts the data to CSV format.

An example execution of the `log2csv-eventcount.awk` is:

```
awk -f log2csv-eventcount.awk \
output/experiments_SD-VBS_stitch_circle_pi4-exp11.log
```

data2linearchart.py

The `data2linearchart.py` script outputs a CSV input file containing a single experiment to a graphical data visualization. In the graphic, the iteration numbers are placed on the `x`-axis and their corresponding cycles data is placed on the `y`-axis. If PMU events data is present for the experiments, it will be plotted together with the cycles data (using a twin axis). See Figure A.2 for an example output image.

An example execution of the `data2linearchart.py` is:

```
python data2linearchart.py \
--input-file=report/data/cyclesdata-core3-configseries211-configbench322-offset10.csv \
--output-directory=report/img \
--maximum-observations=250 --movingaverage-window=0 \
--process-events=True
```

[7] http://pgfplots.net

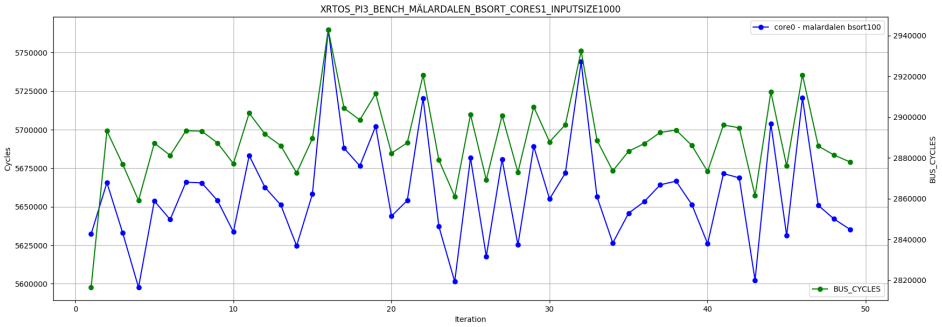


Figure A.2: **Example chart of repeated iterations and measured CPU cycles and bus cycles. The blue line represents the cycles data of the Mälardalen `bsort` benchmark, the green line represents the number of bus cycles measured on the same core on which the task was run.**

The options of the `data2linearchart.py` script are:

- `--input-file` — Path and filename of the input CSV file containing the experiment cycles data.
- `--output-file` — Path of the directory where to place the output PNG files.
- `--maximum-observations` — Maximum number of observations to include in the output plot.
- `--movingaverage-window` — Size of the moving average window to plot, instead of the actual cycles data. The default is a moving average window of 0, which means do not plot the moving average.
- `--process-events` — Whether or not to process the events data, by default the events data are processed.
- `-v, --verbosity` — Define the verbosity for the program, which can be either CRITICAL, ERROR, WARNING, INFO or DEBUG. By default, the verbosity level is set to INFO.

log2data_and_summaries.py

The `log2data_and_summaries.py` script takes the CSV files which are generated by the `awk` scripts, and splits these files into files containing single experiments. Several output options are possible, which are described below.
An example execution of the `log2data_and_summaries.py` script is:

```
python log2data_and_summaries.py \
  --input-file=output/experiments_Mälardalen_matmult_circle_pi4-2-cycles.csv \
  --output-directory=report/data --output-mode=data --metric=cycle
```

The options of the `log2data_and_summaries.py` script are:

- `--input-file` — Path and filename of the CSV input file containing the cycles data or events data, where several experiments (may be) combined in one file.
- `--output-directory` — Path of the output directory, to where the output CSV files containing single experiments must be written.
- `--output-mode` — The output mode determines whether an aggregated `summary` of the experiment must be generated, or whether the `data` must be written to the output file.
- `--metric` — Whether the `cycles` are to be converted to a single experiment file, or `data` are to be converted to a single file. This option can only work for output-mode equal to `data`.
- `-v, --verbosity` — Define the verbosity for the program, which can be either CRITICAL, ERROR, WARNING, INFO or DEBUG. By default, the verbosity level is set to INFO.

slowdown_factors.py

The `slowdown_factors.py` script computes the slowdown factors for the experiments. For this, the experiment definition Excel file is read, to be able to match the experiment with co-runners to its counterpart without co-runners.

An example run of the `slowdown_factors.py` script is:

```
python slowdown_factors.py \
--input-file xlsx/experiments_SD-VBS_stitch_circle_pi4.xlsx \
--output-file slowdown-factors-sdvbs-stitch_pi4-20201019.csv \
--csv-file-prefix=experiments_SD-VBS_stitch_circle_pi4
```

The options of the `slowdown_factors.py` script are:

- `--input-file` — Path and filename of the input Excel file containing the experiment definitions.
- `--output-file` — Path and filename of the CSV output file.
- `--csv-dir` — Path of the directory where the input CSV files are stored, which contain the log data to be analyzed.
- `--csv-file-prefix` — Prefix of the input CSV filenames to be analyzed. The prefix acts as a filter, to make the script not read non-relevant CSV files.
- `--data-dir` — Path of the directory where the data files are stored. This directory contains the CSV files which were already separated per experiment.
- `-v, --verbosity` — Define the verbosity for the program, which can be either CRITICAL, ERROR, WARNING, INFO or DEBUG. By default, the verbosity level is set to INFO.

maketex-cyclessummaries.m4

A separate step is the generation of a PDF report containing graphical output of the aggregated cycles summary information. The generation is done using LaTeX templates, which are in itself generated by the `maketex-cyclessummaries.m4` script.

An example execution of the `maketex-cyclessummaries.m4` script is:

```
m4 -Dfilename=data/cyclessummary-DISPARITY_CORES4_INPUTSIZE32.csv \
-Dconfig_series=3111 -Dconfig_benchmarks= \
-Dlabel=DISPARITY_CORES4_INPUTSIZE32 maketex-cyclesummaries.m4
```

Normally, the `maketex-cyclessummaries.m4` script is executed by the `Makefile` which was mentioned above. Another `Makefile`, which is located in the `run-co-runners/experiment/report` directory, generates the final PDF containing the PGFPlots figures.

A.4.3 Jupyter notebook files

Jupyter[8] is a web application in which users can create documents ('notebooks') in which code (e.g. Python) is mixed with documentation. The code within the notebook can be executed to generate output, such as data visualizations. In this project, the `jupytext`[9] extension has been used for an automatic conversion of the Jupyter notebooks to Python scripts, which can be committed to Git.

Several Jupyter notebook files have been created, in which output data is read and transformed to various data visualizations. These notebooks are part of the `run-co-runners` Git repository, and serve as examples of how the log data can be transformed to more meaningful information on the experiments. Please note that the `jupytext` extension is needed to convert the example notebooks from Python to the Jupyter notebook format (with `.ipynb` extension).

The example Jupyter notebooks are:

- `notebook_mälardalen-bsort-pi3.py` — This notebook contains the results of the experiments with the Mälardalen `bsort` benchmark. It contains boxplot visualizations of the experiments with varying input data sizes, and the computation of the Mann-Whitney U hypothesis tests.

- `notebook_sdvbs-stitch-pi4.py` — This notebook reports on the results of the SD-VBS `stitch` benchmark. It reads the slowdown factors from the CSV file `slowdown-factors-sdvbs-stitcho_pi4-20201019.csv`, and prints `pandoc` data frames from the data. A data visualization is generated containing slowdown effects in relation to the size of the delayed execution.

- `notebook_sdvbs-disparity-pi3.py` — This notebook features various data visualizations from the experiments with the SD-VBS `disparity` benchmark. Slowdown factors are read from the corresponding CSV file and printed in the form of data frames. Several output graphics are created by the use of matplotlib.

[8] https://jupyter.org
[9] https://jupytext.readthedocs.io

- `notebook_sdvbs-disparity-pi4.py` — This notebook contains the same data visualizations like the above notebook, but for the experiments that were run on the Raspberry Pi 4.
- `notebook_mälardalen-matmult-pi3.py` — This notebook contains data visualizations of experiments with the Mälardalen `matmult` benchmark. Slowdown factors are printed and visualizations are created showing the effects of delayed execution of the co-runners.
- `notebook_mälardalen-matmult-pi4.py` — This notebook contains the same data visualizations as the notebook described above, but for the experiments that were run on the Raspberry Pi 4.

A.4.4 Known limitations and gotchas

In the following, some known limitations of our tool are reported.

- System freezes — In rare cases, a core running a task would hang. The cause for this behavior has not been found. The environment in which the tasks run is difficult to debug, because of multiple cores running simultaneously in a bare metal environment. The `run_experiments.py` script works around this problem, by resetting the Raspberry Pi upon a timeout. This timeout is generated when no data has been received from the Raspberry Pi for a long period of time.
- Booting the Raspberry Pi — Sometimes, the Raspberry Pi 3 would not boot from the network. Again, the `run_experiments.py` script uses the timeout mechanism to detect a failure to boot, and will reset the Raspberry Pi if no data is not received for a long period of time.
- Users should be aware of the fact that, to be able to automatically reset the Raspberry Pi, some header pins may need to be soldered on the RUN header of the Raspberry Pi.
- While testing on multiple computers, sometimes the virtualized TFTP server could not be used to boot from the virtual network created by VirtualBox. A solution is to place the TFTP server in the local LAN.