5

Timing Analysis Methodology

Vincent Nélis, Patrick Meumeu Yomsi and Luís Miguel Pinho

CISTER Research Centre, Polytechnic Institute of Porto, Portugal

This chapter focuses on the analysis of the timing behavior of software applications that expose real-time (RT) requirements. The state-of-the-art methodologies to timing analysis of software programs are generally split into four categories, referred to as *static*, *measurement-based*, *hybrid*, and *probabilistic analysis* techniques. First, we present an overview of each of these methodologies and discuss their advantages and disadvantages. Next, we explain the choices made by our proposed methodology in Section 5.2 and present the details of the solution in Section 5.3. Finally, we conclude the chapter in Section 5.4 with a summary.

5.1 Introduction

Most of the timing analysis tools focus only on determining an upper-bound on the Worst-Case Execution Time (WCET) of a program or function code that runs in isolation and without interruption. In other words, these tools do not consider all the interferences that the execution of the analyzed code may suffer when it runs concurrently with other tasks or programs on the same hardware platform. They typically ignore all execution interferences due to the contention for shared software resources (e.g., data shared between several tasks) and shared hardware resources (e.g., shared interconnection network)[1] [1]. Interferences from the operating system (OS) which frequently re-schedules and interrupts the programs are also ignored by WCET analyzers. All these interactions between the analyzed task, the OS, and all the

[1]Note that the OTAWA timing analysis tool is able to analyze parallel code with synchronization primitives [1].

other tasks running in the system are assessed separately and sometimes they are incorporated into a higher-level schedulability analysis. For the timing requirements to be fulfilled, it is neither acceptable nor realistic to ignore these sources of contention and interference at the schedulability-analysis level.

WCET analysis can be performed in a number of ways using different tools, but the main methodologies employed can be classified into four categories:

1. Static analysis techniques
2. Measurement-based analysis techniques
3. Hybrid analysis techniques
4. Measurement-based probabilistic analysis techniques

Note that the first three methodologies are usually acknowledged as equally important and efficient as they target different types of applications. In addition, they are not comparable in the sense that one technique has not been proven to dominate the others. The fourth technique is more recent and thus fewer results are available.

Measurement-based techniques are suitable for software that is less time-critical and for which the average-case behavior (or a rough WCET estimate) is more meaningful or relevant than an accurate estimate like, for example, in systems where the worst-case scenario is extremely unlikely to occur. For highly time-critical software, where every possible execution scenario must be covered and analyzed, the WCET estimate must be as reliable as possible and *static* or *hybrid* methods are therefore more appropriate. *Measurement-based probabilistic analysis* techniques are also designed for safety-critical systems to derive safe estimated execution time bounds, but they are not yet sufficiently mature to report on their efficiency and applicability. Indeed, a consensus is still to be reached in the research community on this matter.

For the execution time of a single *sequential* program run *in isolation*, Figure 5.1 shows how different timing estimates relate to the WCET and best-case execution time (BCET). The example program has a variable execution time that depends on (1) its input parameters and (2) its interactions with the system resources. The darker curve shows the actual probability distribution of its execution time; its minimum and maximum are the BCET and WCET respectively. The lower grey curve shows the set of execution times that have been observed and measured during simulations, which is a subset of all executions; its minimum and maximum are the *minimal measured time* and *maximal measured time*, respectively. For both static analysis tools

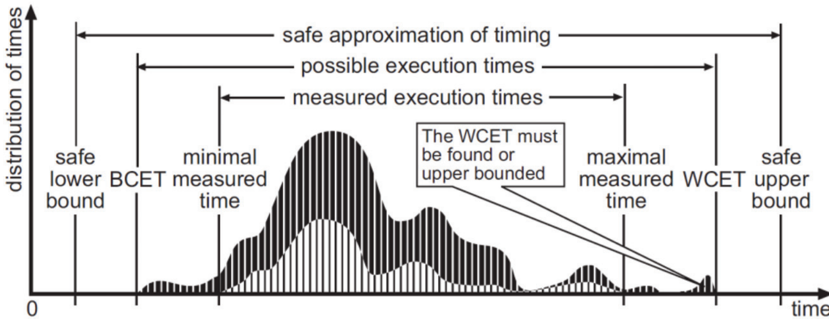

Figure 5.1 Example distribution of execution time (picture taken from [2]).

and measurements-based tools, in most cases the program state space and the hardware complexity are too large to exhaustively explore all possible execution scenarios of the program. This means that *the measured times are likely to be optimistic* and the *estimated times are likely to be pessimistic* – i.e., the measured times will in many cases overestimate the actual BCET and underestimate the actual WCET, while the approximated estimated times will in many cases underestimate the actual BCET and overestimate the actual WCET.

The next four subsections introduce each of the four timing-analysis methodologies and discuss their potential advantages and disadvantages.

5.1.1 Static WCET Analysis Techniques

Static WCET analysis is usually performed in three conceptual and possibly overlapping phases.

1. A flow analysis phase in which information about the possible program execution paths is derived. This step builds a control flow-graph from the given program with the aim of identifying the worst path (in terms of execution time).
2. A low-level analysis phase during which information about the execution time of atomic parts of the code (e.g., instructions, basic blocks, or larger code sections) is obtained from a model of the target architecture.
3. A final calculation phase in which the derived flow and timing information are combined into a resulting WCET estimate.

Flow analysis mostly focuses on loop bound analyses, hence upper-bounds on the number of iterations in each looping structure must be known to derive WCET estimates. Similarly, recursion depth must also be bounded.

Automatic methods to find these bounds have been proposed by the research community but for many available tools, some annotations on the maximum number of iterations in a loop must be provided manually in the code of the tasks by the application developer. Another purpose of flow analysis is to identify infeasible execution paths, which are paths that are executable according to the control-flow graph but are not feasible when considering the semantics of the program and the possible input data values. Discarding unfeasible paths at an early stage of the analysis considerably reduces the search space when trying to identify the longest path.

Low-level analysis methods typically use models of all the hardware components and their arbitration policies, including CPU caches, cache replacement policies, write policies, instruction pipeline, memory bus and their arbitration policies, etc. These models are typically expressed in the form of complex mathematical abstractions for which a worst-case operation can be estimated.

Pros: There are a few advantages of using static analysis techniques that rely on mathematical models.

- It eliminates the need for having the actual hardware available, which removes the cost of acquiring and setting up the target platform.
- It enables safe WCET upper-bounds to be derived without running the program on the target platform while still considering the influence of the state changes in the underlying hardware [3]. State changes include, e.g., a cache line being evicted, a pipeline being totally flushed out, etc.

Cons: On the downside, we shall note the following drawbacks.

- These approaches rely heavily on having an accurate model of the timing behavior of all the target hardware components and management policies, including modeling features like pipelines and caches that substantially affect the execution time of the task being executed. Although the embedded market used to be traditionally dominated by simple and predictable processors (which used to be moderately "easy" to model and allowed for deriving safe and tight bounds), with the increased computational needs of modern embedded systems, designers have moved to more complex processors which are now mainly designed for performance and not for predictability. For this new generation of processors, designing an accurate hardware model is very challenging, as all the intricacies contributing to the variation in the task execution times (e.g., caches, pipelines, out-of-order execution, branch prediction,

automatic hardware prefetching, etc.) should be captured by the model to provide safe and sufficiently tight bounds. Because it is hardly feasible to accurately model all these acceleration mechanisms and their operation, static methods typically forbid their use and are struggling to adapt to modern hardware architectures.

- Besides the difficulty of modeling all these performance-enhancement hardware features, it must also be noted that generally, chip manufacturers do not publish the details of their internal workings, which further complicates/makes impossible the design of an accurate model.
- Although static approaches have the advantage of providing safe WCET bounds, they can be very pessimistic at times. This is because generally, each hardware resource is modeled separately, and all the worst-case estimates are then composed together to form the final WCET bound. However, at runtime, it is often impossible for all these individual worst-case scenarios to happen at the same time.
- The hardware model must be thoroughly verified to ensure that it indeed reflects the target hardware; failing to capture inherent performance enhancing features may result in overestimations of the execution times, whereas capturing all system states in a complex machine may lead to unacceptably long analysis times. Building and verifying the timing model for each processor variant is expensive, time consuming, and error prone. Custom variants and different versions of processors often have subtly different timing behaviors, rendering timing models either incorrect or unavailable.

It is very important to stress at this point that static analysis techniques have been designed primarily to analyze simple software codes meant to run on simple and predictable hardware architectures. These targeted codes are typically implemented by using high-level programming languages and by obeying strict and specific coding rules to reduce the likelihood of programmer error.

The modeling framework adopted by static analysis lends itself to formal proofs which help in establishing whether the obtained results are safe. Today, there are several static WCET tools that are commercially available, including *aiT* [4] and *Bound-T* [5]. Note that Bound-T is no longer actively developed due to both commercial and technical reasons. We redirect the interested reader to their website (http://www.bound-t.com/) for further details on this matter. There also exist several research prototypes, including *Chronos* [6], developed at National University of Singapore, *Heptane* [7], developed at

the French National Institute for Research in Computer Science and Control (INRIA) IRISA in France, *SWEET* [8], developed at Mälardalen Real-Time Research Center (MRTC) in Sweden, and OTAWA [9] from IRIT in France.

5.1.2 Measurement-based WCET Analysis Techniques

The traditional and most common method in the industry to determine program timing is by measurements. The basic principle of this method follows the mantra that "the processor is the best hardware model." The program is executed many times on the actual hardware, with different inputs and *in isolation*, and the execution time is measured for each run by instrumenting the source code at different points [10]. Each measurement run exercises only one execution path throughout the program, and thus for the same set of input values, several thousands of program runs must be carried out to capture variations in execution time due to the fluctuation in system states. For those measurement-based approaches, the main challenge is essentially to identify the set of input arguments of the application that leads to its WCET.

Pros:

- Measurements are often immediately at the disposal of the programmer, and are useful mainly when the average case-timing behavior or an approximate WCET value is of interest.
- Most types of measurements have the advantage of being performed on the actual hardware, which avoids the need to construct a hardware model and hence reduces the overall cost of deriving the estimates.

Cons:

- Measurements require that hardware is available, which might not be the case for systems for which the hardware is developed in parallel with the software.
- It may be problematic to set up an environment which acts like the final system.
- The integrity of the actual code to be deployed in the target hardware is somehow depleted by the addition of the intrusive instrumentation code to measure the time, i.e., the measurements themselves add to the execution time of the analyzed program. This problem can be reduced, e.g., by using hardware measurement tools with no or very small intrusiveness, or by simply letting the added measurement code (and thus the extra execution time) remain in the final program. When doing measurements,

possible disturbances, e.g., interrupts, also have to be identified and compensated for.

- For most programs, the number of possible execution paths is too large to do exhaustive testing and therefore, measurements are carried out only for a subset of the possible input values, e.g., by giving potential "nasty" inputs which are likely to provoke the WCET, based on some manual inspection of the code. Unfortunately, the measured times will in many cases underestimate the WCET, especially when complex software and/or hardware are being analyzed. To compensate for this, it is common to add a safety margin to the worst-case measured timing, in the hope that the actual WCET lies below the resulting WCET estimate. The main issue is whether the extra safety margin provably provides a safe bound, since it is based on some informed estimates. A very high margin will result in resource over-dimensioning, leading to very low utilization while a small margin could lead to an unsafe system.

5.1.3 Hybrid WCET Techniques

Hybrid approaches, as the name implies, present the advantages of both static and measurement-based analysis techniques. Firstly, they borrow the flow-analysis phase from static methods to construct a control flow-graph of the given program and identify a set of feasible and potentially worst execution paths (in terms of execution time). Next, unlike static methods that use mathematical models of the hardware components, hybrid tools borrow their second phase from measurement-based techniques and determine the execution time of those paths by executing the application on the target hardware platform or by cycle-accurate simulators. To do so, the source code of the application is instrumented with expressions (instrumentation points) that indicate that a specific section of code has been executed. These instrumentation points are typically placed along the paths identified in the first phase as leading to a WCET. The application is then executed on the target hardware platform or on the simulator to collect execution traces. These traces are a sequence of time-stamped values that show which parts of the application has been executed. Finally, hybrid tools produce performance metrics for each part of the executed code and, by using the performance data and knowledge of the code structure, they estimate the WCET of the program.

Pros:

- Hybrid approaches do not rely on complex abstract models of the hardware architecture.

- They generally provide safe WCET estimates (i.e., higher than the actual WCET) and those are very often tighter than the estimates returned by static approaches (i.e., closer to the actual WCET).

Cons:

- The uncertainty of covering the worst-case behavior by the measurement remains since it cannot be guaranteed that the maximum interference and the worst-case execution scenario has been experienced when collecting the traces during the second phase.
- It is required to instrument the application source code, which poses the same issue of intrusiveness as in measurement-based approaches. Example tools include Rapitime [11] and MTime [12].

5.1.4 Measurement-based Probabilistic Techniques

With the current hardware designs, the execution time of a given application depends on the states of the hardware components, and those states depend in turn on what has been executed previously. A classic example of such a tight relationship between the application and the underlying hardware architecture is the execution time discrepancy that can be observed when a program executes on a processor equipped with a cache subsystem. During the first execution of the program, every request to fetch instructions and data results in a cache miss and must be loaded from the main memory. At the second execution, this information is already in the cache and need not be reloaded from the memory, which results in an execution time considerably shorter than during the first run. Because of this dependence to past events, the set of measured execution times of the same program cannot be seen as a set of IID (independent and identically distributed) random variables and most statistical tools cannot be applied to analyze the collected execution traces.

The objective of measurement-based probabilistic techniques is to break this dependence on past events, so that one can sample the execution behavior of an application and then derive from the sample probabilistic estimates (of any parameter) that apply to its overall behavior, under all circumstances and in all situations. To achieve this goal, researchers are nowadays working on modifying the hardware components and their arbitration policies to make them behave in a stochastic manner, without losing too much of their performance. For example, by replacing the traditional Least Recently Used (LRU) or Pseudo-LRU (PLRU) cache-replacement policy for a policy that randomly chooses the cache line to be evicted (and assuming that every cache

line has the same probability of getting evicted), the time overhead due to cache penalties and cache line evictions can be analyzed as an IID random variable with a known distribution. If every source of interference exhibits a randomized behavior with a known distribution, then the execution time itself can be analyzed statically.

The current trend in probabilistic approaches is to apply results from the extreme value theory (EVT) framework to the WCET estimation problem [12, 13]. In a nutshell, these EVT-based solutions first sample the execution time of an application by running it over multiple sets of input arguments on a randomized architecture that is designed to confer a stochastic behavior on the application runtime. Then, these EVT-based solutions organize the sample into multiple groups/intervals, analyze the distribution of the local maxima within these intervals and then estimate how far the execution time may deviate from the average of that "distribution of the extremes."

Although considerably new, measurement-based probabilistic techniques have been the object of tremendous research efforts in the last few years, most of the breakthroughs in that discipline have been made in the scope of the European projects PROARTIS [14] and PROXIMA [15].

Pros:

- Provide safe and potentially tighter WCET estimates than static and hybrid techniques.
- Provide information not only on the WCET of a program but on the complete spectrum of the distribution of its execution time.

Cons:

- Require modifying the hardware to ensure that the components exhibit a stochastic behavior.
- As the IID requirement is hardly verified in currently available platforms (especially COTS platforms), the applicability of measurement-based probabilistic techniques is limited.

5.2 Our Choice of Methodology for WCET Estimation

As seen in the previous section, there exist several methodologies to estimate the WCET of an application, each with its own advantages and disadvantages. Those methodologies fall into the following main categories, namely static, measurement-based, and hybrid. Here we would like to briefly re-iterate on why among those four methodologies we decided to use a measurement-based approach.

There is currently an evident clash of opinions in the research community about which methodology prevails over the others. During the last two years we had the opportunity to debate with partisans of each of these approaches. It is important to stress that we do not mean to take a side in this book, simply because we recognize that each approach comes with its own set of strengths and weaknesses. Our methodology simply uses the one whose downsides impede as little as possible our objectives. The following subsections summarize our opinion on the matter and present the observations that have driven our choice towards using a measurement-based technique.

5.2.1 Why Not Use Static Approaches?

In this section, we present some of the reasons why we did not choose static approaches to timing analysis, but rather opted for a measurement-based approach. Before going into the details, it is worth mentioning that recent COTS manycore platforms present complex and sophisticated architectures such that it is very challenging at design time, if not impossible, to come up with an accurate model for all the behavioral implications associated with the possible operational decisions that the system can take at runtime. This claim holds true even for the most experienced systems designers.

> *By using hardware platforms such as the Kalray MPPA-256, or any other platform designed to provide high performance, we argue that it is practically infeasible to derive WCET estimates by using static timing analysis techniques.*

In theory, it is always possible to extract safe and reliable timing models and define mathematical abstractions to study the behavior of a deterministic system. However, we argue that it is practically challenging to define and use static mathematical models of the considered platforms, mainly because of:

The inherent system complexity: Typical COTS hardware components are extremely complex. Currently the market of embedded and electronic components is unarguably driven by the ever-increasing need for higher performance. The only way to constantly enhance the performance is to optimize the produced chips and boards by adding all sorts of optimization features. Optimization is achieved by allowing the system to take and revise its operational decisions on-the-fly, at runtime, based on the current workload of the system or any informational data collected about the running application and its environment. Since those decisions are taken at runtime, it is impossible to predict the exact behavior of the system at the analysis time.

The only option for static tools is to assume that the system will most of the time be in a worst-case situation, in which the optimization features will have very little or no effect. This makes static models pessimistic and the produced timing estimates may not reflect accurately the actual timing behavior of the system.

The human resources required: An increased system complexity leads to a longer time-to-model. Developing a draft model of a platform may take up to several years to reach the desired level of accuracy and be validated. Besides this fact, our software stack and methodology aim at being platform agnostic and therefore be applicable to a large set of hardware platforms. To this end, they should provide a generic abstraction between the application logic and the system interfaces so that the development costs and efforts are always reasonable and limited. This is an objective for which the inherent portability of measurement-based solutions appears to be more appropriate.

Portability: The "rigidity" of static approaches: Using static timing analysis techniques goes against our goal of developing a flexible and generic framework which can be "easily" ported to different platforms from various vendors. This has been a key driver in the development of our timing analysis methods, in order to increase the exploitation opportunities in multiple application domains.

The non-availability of the specification details: To devise accurate models, all the information about the target platform must be available and accurate. This is not the case in practice. Chip manufacturer generally keep most information secret, unfortunately.

The complexity of the execution environment: Static timing analysis tools are designed primarily to focus on applications executed *sequentially* in safety-critical embedded systems. Those systems generally provide a very time-predictable and "inflexible" runtime environment in which every mapping and scheduling decision is statically taken at design-time and is then final. Unlike those systems, the software stack considered in this book offers a much more complex and dynamic runtime environment composed of multiple conceptual layers: the code of the RT tasks is executed *in parallel* by being fractioned into OpenMP tasks, those tasks are mapped to clusters, then to threads inside the clusters, and then these threads are scheduled statically or dynamically on the cores. The dynamicity of the processor resource usage ensures a decent application throughput (by maximizing the utilization of

the available computing resources) but it naturally impacts adversely on its time-predictability.

> *Traditional hybrid approaches are also not applicable as the complexity of the software stack makes the static control-flow analysis step impossible.*

Since the RT tasks execute in parallel, and even using static mapping approaches, the total order of execution of task-parts is only determined at runtime, it is thus infeasible to investigate all possible scenarios at design-time to identify the worst-case execution flow/path. It is important to re-iterate that traditional timing analysis techniques have been designed primarily to analyze "simple" software codes executed on "simple" and predictable hardware architectures, typically implemented by using low-level programming languages and by obeying strict and specific coding rules to reduce programmer's errors. The framework presented in this book clearly targets much more complex software applications that exhibit a high degree of flexibility and dynamicity in their execution.

5.2.2 Why Use Measurement-based Techniques?

> *In measurement-based approaches, WCET estimations are derived from values that have been observed during the experimentation. What about the values that have not been observed? How can we account for them and be sure that the WCET estimates are reliable?*

Critics of measurement-based approaches for estimating the WCET of an application make a simple yet very valid point. The actual WCET is unknown and is very likely not to be experienced during testing. Even worse, it is not even possible to know whether the worst case has been observed or not. In short, this means that there is no guarantee that such an approach can forecast the exact value of the WCET. All measurement-based techniques implicitly infer a WCET from values for which the "distance" from the actual worst-case is unknown. A direct consequence is that, although those techniques make predictions based on sophisticated and elaborate computations, formally speaking, they can never guarantee that their predictions are 100% "safe". This may be problematic for applications requiring hard RT guarantees, typically in safety-critical systems for instance.

However, one can note that in many application domains, certifiable guarantees based on unquestionable and provable arguments are not required.

For instance, many applications need only "reliable" estimations, in the sense that one must be able to rely on those values and measure the risk of them being wrong (through confidence levels provided by the analysis, for example).

Estimations of the trustworthiness of the produced values (i.e., the confidence in those values) can be expressed through probabilities derived by statistical tools. Specifically, in our approach, the traces of execution times collected at runtime are fed into a statistical framework, called DiagXtrm, in which they are subjected to a set of tests to verify basic statistic hypotheses, such as stationarity, independence, extremal independence, execution patterns/modes, etc. Depending on the results of those tests, it is determined whether the EVT can be applied to those traces. If the tests are successful, the EVT is used to "extrapolate" the recorded execution times and accurately identify the higher values that have not been observed during testing, but for which the likelihood of occurrence is not statistically impossible. Besides this, our framework also provides techniques to assess how "trustworthy" those EVT estimations really are. This last step is of fundamental importance to evaluate the quality of the estimations and find out whether confidence can be placed into the analysis.

Despite all the interesting features provided by the application of EVT to the WCET determination problem, it has been widely criticized in the research community. The main argument against it is that the process of creating the traces (i.e., the execution of an application's code by a given hardware platform) is known to be a process which is neither independent nor identically distributed, which is a prerequisite to the application of the EVT to a data sample. We believe that this argument, although correct because the process is *de facto* not inherently IID, does not allow to conclude on the non-applicability of the EVT. In our view, being an IID process is not necessary, provided that the said process behaves as if it were. This is why the EVT has been applied in so many application domains where it is today recognized to provide helpful and satisfactory results. EVT is used for instance to predict the probability distribution of the amount of large insurance losses, day-to-day market risk, and large wildfires. Needless to say, none of these processes are truly IID.

Whether this is right or not is disputable and we do not intend to close the discussion in this chapter. However, we believe that the doubt this casts on the applicability of the EVT makes this framework worth being investigated further and hopefully will unveil its true potential. In case we are wrong, we will hopefully discover why it is not applicable and close the debate that has been going on already for several years.

> *In measurement-based approaches, the integrity of the actual code to be deployed in the target hardware is somehow depleted by the addition of the intrusive instrumentation code to measure the time; in other words, the measurements themselves add an overhead to the execution time of the analyzed program.*

This problem can be reduced, e.g., by using hardware measurement tools with no or very small intrusiveness, or by simply letting the added measurement code (and thus the extra execution time) remain in the final program. When doing this, possible disturbances like interrupts also have to be identified and compensated for. The intrusiveness of the instrumentation code is discussed in Section 5.3.5 and we provide efficient solutions to deal with it.

Nearly all the embedded platforms, like the MPPA-256 platform considered in our experimentations, provide a lightweight and non-intrusive trace system that enables the collection of execution traces in predefined time bounds. By using this trace system, we are able to collect meaningful traces of execution without generating too many disturbances in the regular timing behavior of the analyzed application. Based on all the experiments conducted on the Kalray board, we concluded that the time necessary to record a time stamp is 52 clock cycles. By placing "trace-points" (points in the program where the current time is recorded) at well-defined places, we can thus easily subtract the overhead associated with measuring the time itself.

Wrapping things up:

The best candidates for the worst-case timing analysis of the type of workloads considered in this book are the measurement-based approaches. Thus, our proposed methodology relies on timing-related data collected by running the application on the target hardware. This way, we avoid both the burden of modeling the various hardware components (which takes considerable effort and time), as in static timing analysis tools; and the pitfalls and pessimism associated with the over-approximations resulting from the confidentiality, and thus the non-availability, of specific information related to the internal configuration of the components. In addition, the fact that our approach is not tied to specific hardware infrastructures and application designs allows it to benefit from a higher flexibility and portability than static timing analysis methods, and it considerably reduces the time-to-model and time-to-result. In the next sections, we will discuss the specifics of our method and how we propose to overcome or at least mitigate the negative aspects inherent to measurement-based techniques.

5.3 Description of Our Timing Analysis Methodology

5.3.1 Intrinsic vs. Extrinsic Execution Times

The execution time of any piece of code, e.g., a basic block, a software function, or an *OpenMP task-part*, can be seen as composed of two main terms: the *intrinsic* execution time spent executing the instructions of the code, and the *stalling* time, i.e., the time spent waiting for a shared software or hardware resource to become available. To understand how timing analysis is performed in this book, it is fundamental to understand the difference between these two components. If the analyzed software function does not have a functional random behavior (i.e., the outcome of evaluating a condition is never the result of an operation involving randomly generated numbers), then any input dataset always produces one output (and this output remains the same no matter how many times the function is executed on the same input). Further, for a given input dataset, the execution path taken throughout the function's code will always be the same. That is, under this assumption of not involving randomness in the control flow of the analyzed function, running it over a given set of input data over and over again always results in executing the exact same sequence of instructions and eventually, it always produces the same output.

For a given input dataset, we call the "*intrinsic* execution time" of a function the time that it takes to produce its output, assuming that all software and hardware services provided by the execution environment and shared among different cores are always available, and thus the core running that function never stalls waiting for one of these resources to become available. That is, the intrinsic execution time of a function is its execution time when it runs in isolation, i.e., with no interference whatsoever with the rest of the system on the shared resources. On a perfectly predictable hardware architecture where every instruction takes a constant number of cycles to execute, running the same function in isolation over the same set of input arguments should always results in the exact same execution time. Although this may sound like a very strong assumption, we will see that on a platform such as the Kalray MPPA-256 this property is satisfied. By running a preliminary set of tests with the same program an arbitrary number of times over the same inputs, we experienced a variation of its execution time of typically less than 0.1% of the maximum observed.

For a given input dataset, we call the "*extrinsic* execution time" of a function the time that it takes to produce its output, assuming a maximum interference on all the shared resources. That is, the extrinsic execution time

of a function is its execution time assuming that all the software and hardware services provided by the execution environment and shared among the cores are constantly saturated by concurrent requests from other system components. Contrary to the intrinsic execution time, on mainstream multicore architectures the extrinsic execution time is subject to huge variabilities due to the high number of processor resources shared amongst software functions.

5.3.2 The Concept of Safety Margins

When testing an application and measuring its execution time, it is very likely, if not certain, that the (usually very rare) situation where the application takes its maximum execution time does not occur. This is due to either of the following reasons:

1. The testing process failed to identify the set of input arguments that takes the longest execution path throughout the program's code, i.e., the path that leads to the WCET.
2. The testing process found the execution path(s) leading to the WCET but did not generate the maximal possible interference while exercising those paths. This means that the actual WCET is not observed only because the interference patterns generated during testing did not put the application into the worst execution conditions.

Regarding the first case, for most programs, the number of possible execution paths (in comparison to the high number of possible inputs) is too large to make exhaustive testing possible and/or realistic. Therefore, measurements are carried out only for a subset of input values. Typically, the testing process starts with the identification of a set of potentially "nasty" inputs that are likely to make the program take the longest execution path throughout its code and provoke its WCET. This step is typically supervised and based on some manual inspection of the code. Note that powerful tools exist such as the Rapita Verification Suite (RVS) that incorporates a code-coverage tool (RapiCover [16]) to test all parts of a given code and guarantee its full coverage during testing. We believe that such tools may be employed to help system designers identify the "worst" input datasets.

The problem of defining the worst input dataset(s) is thus not new, and to some extent it is independent of the underlying hardware architecture. Of course, the execution time of a given path depends on the execution time of each instruction in that path, and therefore is dependent on the architecture, but the method to search the space of all possible inputs and identify those that

lead to the longest execution path is platform-agnostic. Since the problem was already there on single-core architectures, with mature solutions for it, we do not focus, in this book, on improving this part of the process.

Regarding the second point, it is always assumed that the worst-case interference is not observed during testing and therefore the maximum execution time recorded is an under-approximation of the actual WCET. To compensate for this, it is common to add a safety margin to the measured WCET, in the hope that the actual WCET lies below the resulting augmented estimation. The main question that remains open is whether the extra safety margin provably provides a safe bound, since it is based on some informed estimates. In principle, a very high margin yields an upper-bound on the execution time that is likely to be safe (i.e., greater than the actual WCET), but results in an over-dimensioned system with a low utilization of its resources, whereas a small margin may lead to an under-estimation of the actual system (worst-case) needs.

Traditionally, the magnitude of the safety margin applied to the maximum measured execution time is based on an estimation of the maximum interference (from the system or from other applications) that has not been observed during the testing phase but that the analyzed application could potentially incur at runtime. For single-core systems, this estimation of the worst-case interference is usually built on past experience. For example, in the IEC 61508 standard [17] related to functional safety of electrical/electronic/programmable electronic safety-related systems, to ensure that the working capacity of the system is sufficient to meet the specified requirements, it is mentioned that:

> *"For simple systems an analytic solution may be sufficient, while for more complex systems some form of simulation may be more appropriate to obtain accurate results. Before detailed modeling, a simpler 'resource budget' check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is infeasible. Even if the design passes this check, performance modeling may show that excessive delays and response times occur due to resource starvation. To avoid this situation, engineers often design systems to use some fraction (for example 50%) of the total resources so that the probability of resource starvation is reduced."*

As explained above, it is a common practice to simply add a margin of 50% (or any other percentage depending on the user's preferences and his level

of confidence in those margins) to the maximum execution time observed. Unfortunately, on multicore and manycore architectures, experts are not yet able to safely estimate reliable margins, as there is no prior experience to be relied upon. Hence, we must build a new body of knowledge and investigate novel approaches to produce reliable timing estimates and margins, and we must motivate these estimations and justify why we believe they are reliable. Our move towards this ambitious goal is described in short in the following subsection.

5.3.3 Our Proposed Timing Methodology at a Glance

In this book, we devised methods to extract both the intrinsic and extrinsic execution times. The overall timing analysis methodology consists of four steps:

Step 1: Extraction of the maximum intrinsic execution time

To measure the maximum intrinsic execution time (MIET), we run the analyzed task sequentially on one core and we configure the execution environment in such a way that no other tasks can interfere with its execution. That is, everything is done to nullify the interference with other applications or with the system itself. This way we put the analyzed task in "ideal" execution conditions in which, in the absence of interference, the time taken to execute its code can be assumed to be due solely to the execution of its instructions (without any stalling time). In these conditions, the task to be analyzed is run multiple times, non-preemptively, over a finite set of input data. These input data have been pre-selected and identified as particularly "nasty", i.e., very likely to make the task take its longest execution path throughout its code and provoke its WCET. We do not elaborate on how to select those inputs.

Step 2: Extraction of the maximum extrinsic execution time

The maximum extrinsic execution time (MEET), on the contrary, is obtained by measuring the time taken to execute the analyzed task in conditions of "extreme" interference. That is, everything is done to maximize the interference with other applications and with the system itself. Measuring the execution time of the analyzed task in those "worst" conditions and over the "worst" input datasets give an estimation of the maximum execution time that the task may experience in the presence of other tasks running concurrently.

Step 3: Extract the execution time after deployment

The MIET and MEET can be considered as lower and upper bounds on the actual WCET of the analyzed task, since they estimate the WCET in conditions of no and extreme interference, respectively. These two estimations are useful to the system designers to understand the impact that tasks may have on each other's timing behavior. For instance, it may be desirable to derive a static mapping of the task-parts to the cores in which the task-parts (the portions of code for which the executions are timed or measured) that are highly sensitive to interference (i.e., the difference between their MEET and MIET is large) are mapped to specific cores in a way that they cannot interfere with each other at runtime.

After taking mapping and scheduling decisions based on the values of the MIET and MEET, these decisions are implemented and the whole system is run in its final configuration. Measures are taken again, this time to estimate the execution time of the tasks in its "final" execution environment, i.e., the environment corresponding to the "after-deployment". Timed traces are recorded like in the previous step and are passed to step 4.

Step 4: Estimate a worst-case execution time

The traces collected in Step 3 reflect the actual execution time of every task-part, and from those their individual WCET can be derived or estimated. The simplest way to proceed is to retain the maximum execution time observed as the actual WCET. For safety purpose, an arbitrary extra "safety margin" can be added to that WCET estimation to make it even safer. The magnitude of the margin depends on how much "safer" the system designers want to be, but we would recommend using a margin that does not exceed the MEETs of the tasks (because the MEETs represent the WCET of the tasks in execution conditions that are unlikely to happen at runtime).

However, instead of arbitrarily choosing a margin, we advocate the use of statistical methods to analyze the traces and make a more "educated" choice driven by mathematical assumptions and computations rather than just a "gut feeling". In this book, we use DiagXtrm, a complete framework to analyze timed traces and derive pWCET estimates.

In the next subsections, we describe every step of our methodology.

5.3.4 Overview of the Application Structure

Before we go to the details, let us briefly recall the type of workloads that we are handling in this book and recap what exactly needs to be measured.

In the considered system model, the application comprises all the software parts of the systems that operate at the user-level and that have been explicitly defined by the user. The application is the software implementation (i.e., the code) of the functionality that the system must deliver to the end-user. It is organized as a collection of RT tasks.

An RT task is a recurrent activity that is a part of the overall system functionality to be delivered to the end-user. Every RT task is implemented and rendered parallelizable using OpenMP 4.5, which supports very sophisticated types of dynamic, fine-grained, and irregular parallelisms.

An RT task is characterized by a software procedure that must carry out a specific operation such as processing data, computing a specific value, sampling a sensor, etc. It is also characterized by a few (user-defined or computed) parameters related to its timing behavior such as its WCET, its period, and its deadline. Every RT task comprises a collection of task regions whose inter-dependencies are captured and modeled by a directed acyclic graph, or DAG.

A task region is defined at runtime by the syntactic boundaries of an OpenMP task construct. For example:

```
#pragma omp task
{
        // The brackets identify the boundaries of the task region
}
```

Hence, hereafter we refer to task regions as *OpenMP tasks*. The OpenMP tasking and acceleration models are described in detail in Chapter 3.

An *OpenMP task-part* (or simply, a task-part) is a non-preemptible portion of an OpenMP task. Specifically, consecutive task scheduling points (TSP) such as the beginning/end of a task construct, the synchronization directives, etc., identify the boundaries of an OpenMP task-part. In the plain OpenMP task scheduler, a running OpenMP task can be suspended at each TSP (not between any two TSPs), and the thread previously running that OpenMP task can be re-scheduled to a different OpenMP task (subject to the task scheduling constraints).

The DAG of task regions can therefore be further expanded to form a typically bigger DAG of task-parts. This new graph of task-parts is called the extended task dependency graph (eTDG) of the RT task. Figure 5.2 shows the eTDG of an example application. Our objective is to annotate every node, i.e., task-part, of the eTDG with an estimation of its WCET and then perform a schedulability analysis of the entire graph to verify that all the end-to-end timing requirements were met.

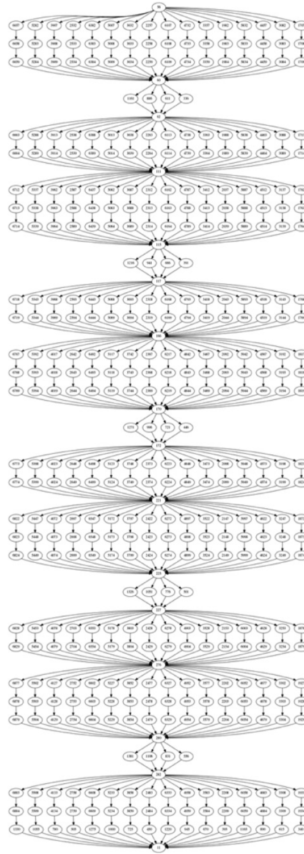

Figure 5.2 Extended task dependency graph (eTDG) of an example application.

5.3.5 Automatic Insertion and Removal of the Trace-points

In this subsection, we discuss how to respectively *insert* (Subsection 5.3.5.1) and *remove* (Subsection 5.3.5.2) trace-points in a given program in an automatic manner.

5.3.5.1 How to insert the trace-points

To measure the execution time of a task-part, we insert a trace-point at its entry and exit points. A trace-point is a call to a system function that records the current timestamp. Therefore, the system will record the time of entering the task-part (i.e., when its execution starts) and the time at which it exits

it; the difference between the two straightforwardly gives the time spent executing the task-part.

Inserting the trace-points into the tasks' code can easily be done by the compiler itself, when creating the executable file. Moreover, upon compiling the code and creating the TDG, the compiler can assign a unique Identifier (ID) to every task-part. Overall, this ID can be used to define a trace-point for the task-part associated with an execution time. For example, using the trace system from the Kalray SDK, we ask the compiler to add the following trace-points at the beginning and end of every task-part as illustrated in the code snippet below:

```
#pragma omp task
{
// The brackets identify the boundaries of the task region
       mppa_tracepoint (  psocrates , taskpartID__in ) ;
       /* code of the task-part */
       mppa_tracepoint (  psocrates , taskpartID__out ) ;
}
```

These trace-points indicate to the Kalray MPPA runtime environment that a time-stamp must be recorded each time the execution meets one of these points (together with the ID of the corresponding task-part). The first argument (here, "psocrates") is the name of the "trace-point provider". The user defines it to help him organize all its trace-points into groups. Informally, it can be thought of as a folder name. The second argument is the name of the trace-point. For every task-part we insert a trace-point called "`taskpartID__in`" at the beginning of the task-part and another trace-point called "`taskpartID__out`" at the end. We do so because the objective of our next tool is to find every matching pair "`*__in/*__out`" of trace-points and compute the difference of timestamps (which naturally corresponds to the execution time of the task-part).

Once all the trace-points are correctly placed into the source code, the compiler must create a separate header file "*tracepoints.h*" in which all the trace-points are declared and then include that file in all source files in which trace-points are used (#include "tracepoints.h").

```
#ifndef _TRACEPOINTS_H_
#define _TRACEPOINTS_H_
#include "mppa_trace.h"

MPPA_DECLARE_TRACEPOINT(psocrates, taskpartID__in,())
MPPA_DECLARE_TRACEPOINT(psocrates, taskpartID__out, ())

... // more trace-points

#endif
```

5.3.5.2 How to remove the trace-points

After the analysis step, when the system is ready to be deployed, it is preferable to remove all the trace-points in order not to leave some "dead code." A code is said to be dead either if it is never executed, or when its execution does not serve any purpose, like for example taking time-stamps and not recording them into a file (which would happen if those trace-points were to be left in the source code when compiling the application to be deployed). However, removing trace-points is not a benign operation.

To illustrate the problem that may arise from removing the trace-points, let us consider the following code.

```
int run_index;
for ( run_index = 0 ; run_index < NB_RUNS ; run_index++ ) {
        mppa_tracepoint(psocrates, main__in);
        user_main();
        mppa_tracepoint(psocrates, main__out);
}
```

The `user_main()` function is a call to the main function of the benchmark program "statemate.c" provided by (15). If we disable all compiler optimizations during the compilation phase (this is important and will play a role later) and run this code 100 times on a single core of a compute cluster of the Kalray MPPA-256, we observe that the execution time oscillates consistently between 88492 and 88497 cycles (see Figure 5.3, left-hand side).

Now, let us add to that code a variable *x* to which we assign an arbitrarily chosen integer (here, 1587) as shown below:

```
int x = 1587;
int run_index;
for ( run_index = 0 ; run_index < NB_RUNS ; run_index++ ) {
        mppa_tracepoint(psocrates, main__in);
        user_main();
        mppa_tracepoint(psocrates, main__out);
}
```

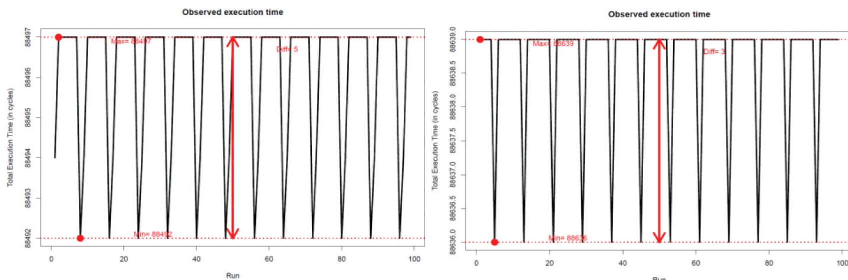

Figure 5.3 Impact of an unused variable on the execution time of an example application.

It is important to stress that the variable x is never used in the program. Since all compiler optimizations are disabled, the variable is not removed from the code by the compiler and is present in the assembly code that it produces. As seen in the Figure 5.3 (right-hand side), the execution time now oscillates consistently between 88, 639 and 88, 636 cycles. This means that the addition of an unused variable to a part of the code which is not even under analysis adds around 140 cycles to the execution time of the measured portion of the code.

This increase in the execution time stems from the fact that after the addition of the line "int x = 1587" to the source code, all subsequent instructions got offset in the system memory by two times the length of an instruction, i.e., the line "int x = 1587" translates to two assembly instructions: one for allocating memory to the variable x and another one for moving the constant "1587" into it. Therefore, the portion of the code being timed has a different "memory layout" as it is mapped to the system memory two "instruction-lengths" further. This in turn impacts on the way the instructions of that part of the code are mapped at runtime to the instruction cache lines and ultimately it results in a perceptive difference in the execution time.

A consequence of this phenomenon is that removing the trace-points after the analysis phase may have for effect to substantially, or at least noticeably, alter the timing behavior of the application and all its task-parts. We came up with two potential solutions to this problem. The simpler one is to leave the trace-points in the code when compiling it for the final release of the application. Although it is a suitable work-around to the memory-shift problem described above, most designers are not in favor of having a dead portion of code, as explained above.

Our second solution is to measure the length, in number of assembly instructions, of the code being executed each time the function `mppa_tracepoint(...)` is called and replace every such call with an equivalent number of NOPs (No Operation assembly instruction). This way neither the semantic of the code nor the memory layout are altered when removing the trace-points. We believe this solution to be both feasible and suitable for use in industrial applications.

5.3.6 Extract the Intrinsic Execution Time: The Isolation Mode

In order to extract the MIET of a task-part, we must start its execution and make sure that it is isolated from the rest of the system. That is, we must nullify all external interference by turning off every other component that could

potentially interfere with (and hence delay) the execution of the analyzed task-part. This is achieved by assigning every task-part of the analyzed real-time task to the same thread, and thus to the same core of the same cluster, and then making sure that all the other cores are kept idle. In other words, under this configuration, the RT task is executed sequentially in a single core. However, the intention of this phase is to analyze the execution time of each task-part in isolation, i.e., without suffering interferences, and not the overall RT task execution time. We call this configuration the *isolation mode*; the real-time task is then said to run *in isolation*.

To setup and enforce this isolation mode, we have implemented a platform-specific API. The current version has been written for the Kalray MPPA-256. The API provides a set of easy-to-use functions to configure the execution environment, as well as a set of global parameters and functions that are used to make sure that:

1. all the openMP tasks are assigned to a single thread,
2. the IO cores and the cluster cores are in sync so that the environment is "sanitized" before and after the execution of every openMP task (nothing runs in the background that could interfere with the execution of the analyzed task), and
3. additional functions allow the user to perform specific operations, either before the runtime, such as deciding the memory-mapping and cache-management policy, or during the runtime, such as invalidating the instruction or data caches before executing each task-part.

The main objective of the API is to create a controlled environment in which every task-part is run over a specific set of inputs and is isolated from the rest of the system so that it incurs minimum interference during its execution.

5.3.7 Extract the Extrinsic Execution Time: The Contention Mode

To extract the MEET of a task-part, we start the task and interfere as much as possible with its execution at runtime. The objective of the contention mode is to create the "worst" execution conditions for the task-parts so that their execution is constantly suspended due to interference with other tasks. In this step, for each task-part, we record the maximum execution time observed under those conditions. This gives us an estimation of the maximum execution time of each task-part when it suffers interference from other tasks on the shared resources.

This *contention mode* is similar to the isolation mode in that all the task-parts of the analyzed real-time task are assigned to the same thread, and thus to the same core within a same cluster, effectively executing the RT task sequentially. However, contrary to the isolation mode that shuts down all the other cores of the cluster (thereby nullifying all possible interference within that cluster), we deploy onto all these other cores small programs called IG, which stands for *Interference Generator*. Those programs are essentially tiny pieces of code that have the sole purpose of saturating all the resources (e.g., interconnection, memory banks) that are shared with the task-parts under analysis. Recall that the objective of the contention mode is to create the worst execution conditions for the execution of the task-parts, conditions in which their execution is slowed down as much as possible due to contention for shared resources.

Implementing the IG that generates the worst possible interference that a task-part could ever suffer is a very challenging, if not impossible, task. This is because the exact behavior of the task-part to be interfered with (i.e., its utilization pattern of every shared resources and the exact time-instants of accessing it) should be known, as well as all the detailed specifications of the platform. Besides, even if that information was known, the execution scenario causing the maximum interference may be impossible to reproduce. Rather than concentrating our efforts on creating such a "worst IG", we have opted for the implementation of an IG that is "bad enough" and used it as a proof of concept to demonstrate how large the time-overhead incurred by the task-parts due to the interference can be.

Our implementation of the IG consists of a single function *IG_main* that is executed by a thread dispatched to every core on which the task-parts are not assigned (recall that the application under analysis is executed sequentially in a single core). That is, every core that is not running the task-parts runs a thread that executes *IG_main*. Essentially, *IG_main* executes three functions, namely:

1. IG_init_inteference_process ()
2. IG_generate_interference ()
3. IG_exit_inteference_process ()

The first one is called upon deploying the IG, at the beginning of *IG_main*, before the task-parts start to execute and be timed. The second one is the main function. It creates interference on the shared resources. The call to that function is encapsulated in a loop that terminates only when the *IG_main* is

```
int* my_array;
inline void IG_init_interference_process()
__attribute__((always_inline));
inline void IG_generate_interference()
__attribute__((always_inline));
inline void IG_exit_interference_process()
__attribute__((always_inline));
```

explicitly told to stop. Finally, the third function is called when all the task-parts have been timed and the analysis process is about to end.

Let us now briefly describe our implementation of the IG on the Kalray MPPA-256. This implementation is provided in a single file, which starts with the declaration of an array of integer called `my_array` and declares the three main functions as described above. The `__attribute__((always_inline))` instruction is used to enforce and oblige the compiler to use inlining for these three methods. The inlining technique is used to waste as little time as possible jumping from one address to another in the code, as jumping does not create interference.

Below is a code snippet of the first function "*IG_init_interference_process().*"

```
inline void IG_init_interference_process() {
       int array_size = 1024;
       // Create an array of Integers. One integer is 4 bytes
       my_array = malloc(array_size * sizeof(int));
       // Fill the array with numbers.
       int cpt = 0;
       for (cpt = 0 ; cpt < array_size ; cpt++) {
              my_array[cpt] = cpt;
       }
}
```

This function simply allocates memory to `my_array` (1024 integers) and fills that memory space with arbitrary values. Note that on the Kalray MPPA-256, a thousand integers occupy roughly half of the private data cache of a VLIW[2] core in a compute cluster.

The third function, "*IG_exit_inteference_process()",* is the simplest as it only frees the memory space held by `my_array` as shown below.

```
inline void IG_exit_interference_process() {
       Free(my_array);
}
```

The second function, "*IG_generate_interference ()*," is the main one and a snippet of its code is presented below.

[2]Very Long Instruction Word.

```
inline void IG_generate_interference() {
        __builtin_k1_dinval();
        __builtin_k1_iinval();
               register int *p = my_array;
        volatile register int var_read;
        var_read = __builtin_k1_lwu(p[0]);
        var_read = __builtin_k1_lwu(p[8]);
        var_read = __builtin_k1_lwu(p[16]);
        var_read = __builtin_k1_lwu(p[24]);
        var_read = __builtin_k1_lwu(p[32]);
        var_read = __builtin_k1_lwu(p[40]);
        var_read = __builtin_k1_lwu(p[48]);
        var_read = __builtin_k1_lwu(p[56]);
        var_read = __builtin_k1_lwu(p[64]);
        var_read = __builtin_k1_lwu(p[72]);
        var_read = __builtin_k1_lwu(p[80]);
        (...)
        var_read = __builtin_k1_lwu(p[1007]);
        var_read = __builtin_k1_lwu(p[1015]);
        var_read = __builtin_k1_lwu(p[1023]);
}
```

The function starts by invalidating the content of the data and instruction caches. Then, it reads every element of "*my_array*", starting from the element K = 0 and moving on iteratively from element K to element ((K+8) mod 1024), until K reaches 1023. This way, every element of the array is read exactly once and every two consecutive readings access data that are located exactly 8 * 4 = 32 bytes apart in the memory (the size of an integer is standard on the Kalray, i.e., 4 bytes). This is done on purpose knowing that the private data cache line of every VLIW core in the compute clusters of the Kalray MPPA-256 is 32 bytes long. Consequently, every reading causes a cache miss and the value must then be fetched from the 2 MB in-cluster shared memory, hence it creates traffic on the shared memory communication channels and potentially interferes with the task-part being analyzed.

By running the task-parts concurrently with these IGs, every request sent by a task-part to read or write a data in the shared memory is very likely to interfere with a read request from one of the IGs. We have conducted experiments on the Kalray MPPA-256 using several use-case applications to evaluate the magnitude of the increase in the execution time due to this interference. Depending on the configuration of the board and the memory footprint of the task-parts and their communication pattern with the memory, the difference between the maximum execution time observed in isolation mode and in contention mode is substantial as the execution time of a task-part may be increased by a factor of 9.

5.3.8 Extract the Execution Time in Real Situation: The Deployment Mode

After determining the intrinsic and extrinsic execution times (i.e., the MIET and the MEET), we communicate them to the mapping and scheduling analysis tools through the annotation of the TDG of the real-time task. Once all necessary mapping and scheduling decisions are taken, the application is run again, but this time in its final production environment. This means that the platform configuration and mapping and scheduling decisions are no longer imposed and defined so as to create specific execution conditions. Then, we collect runtime-timed traces of the task-parts in their final environment, without any supervision or any attempt to explicitly favor or curb the execution of the application.

5.3.9 Derive WCET Estimates

As already discussed, the traces collected in the previous step reflect the actual execution time of every task-part when they run in their final environment, under different execution conditions. The objective of this final step is to derive WCET estimates from those traces. The simplest solution is to retain the maximum execution time observed during the deployment mode as the actual WCET and, for safety purposes, add an arbitrary "safety margin" to that maximum to make it "safer". The magnitude of the margin depends on how much "safer" the system designers want to be, but we would recommend using a margin that does not exceed the MEET. However, instead of arbitrarily choosing a margin, we advocate the use of statistical methods to analyze the traces and make a better thought out choice.

The objective of Measurement-Based Probabilistic Timing Analysis (MBPTA) approaches is to characterize the variability in the execution time of a program through probability distributions and in particular, they aim at deriving probabilistic WCET estimates, a.k.a. pWCET. A pWCET is a probability distribution of the WCET of a program. That is, through MBPTA, the WCET is no longer expressed as a single value but as a range of values, each assigned to a given probability of occurrence with the obvious relation: the higher the value assumed to be the WCET, the lower its probability of occurrence. Based on this framework system designers are in a position to somewhat decide on the reliability of the final WCET estimation, simply by ignoring all values for which the probability of observing an execution time greater than those exceeds a pre-decided threshold. The EVT is a popular theoretical tool used by most MBPTA approaches. The EVT aims at modeling

and estimating better the tail of a statistical distribution, which is *de facto* what the MBPTA is trying to achieve when focusing on the pWCET.

Researchers at the French Aerospace Lab (ONERA), in France, recently proposed a remarkable framework and tool to analyze timed traces and derive pWCET estimates. The framework is called DiagXtrm [18] and defines a methodology composed of three main steps:

1. Analyze the traces
2. Derive pWCET estimates using the EVT
3. Assess the quality of the estimations.

Together with the theory and the definition of the methodology, they developed a tool to diagnose execution time traces and derive safe pWCET estimates using the EVT. However, the EVT can be applied to a given trace only if some hypotheses are verified. Testing those hypotheses is the focus of the first step ("Analysis of the traces") above.

In a nutshell, for safely applying the EVT and getting reliable pWCET estimates, one has to check a few hypotheses including for instance stationarity, short-range dependence, and extreme independence. The *stationarity* of a trace reveals whether measurements belong to the same probabilistic law without knowing it. The *independence* (short-ranged or between the extremes) analysis aims at determining whether there are obvious correlations within the measurements. Systemic effects in a modern hardware platform are so complex and numerous that it is quite impossible to infer the probability of happening of an execution time knowing the value of the preceding ones, i.e., the execution time of an application cannot be inferred from the execution times of its previous executions. System non-determinism, coming from the considered system's degree of abstraction, knowledge, and randomness observed in a timed trace motivate the independence of the measurements that has to be studied at "different scales" (i.e., short-range independences and independences of the extremes). DiagXtrm implements the most advanced tests to verify the stationarity hypothesis and measure the degree of correlation between patterns of different lengths within a trace. Thus, it studies both short-range and distant dependencies between the measurements.

If all the hypotheses are verified, then the EVT is applied to produce pWCET estimates. These estimates are the result of sophisticated computations based on parameters that must be carefully set. The user is in charge of setting those parameters as he wants, and thus has a great influence on the pWCET estimation process. Note however that the DiagXtrm tool provides helpful functions to guide the choice of many of those input parameters.

Finally, the tool features a set of tests to evaluate the quality of the produced estimates, together with other tests to assess the confidence that all the hypotheses were verified. We believe that this last phase is fundamental and is a first step towards building confidence and assessing the reliability of the pWCET estimates.

5.4 Summary

The analysis of the timing behavior of software applications that expose real-time requirements and dedicated to execute on the recent COTS manycore platforms such as the Kalray MPPA-256 raises a number of important issues. Because a reliable and tight WCET estimation for each task running on such a platform is a crucial input at the schedulability analysis level, we showed that it is neither acceptable nor realistic to ignore all the interactions between each analyzed task, the OS, and all the other tasks running in the system. Then, depending of the type of workload that is considered, we also showed that the choice of the methodology to be adopted must be conducted with care. In this chapter, after presenting an overview of all the possible methodologies, and after discussing their advantages and disadvantages, we opted for a measurement-based approach. We explained and motivated this choice and finally presented the details of our solution. Here, we showed that both the intrinsic (MIET) and extrinsic (MEET) execution times of each task are pivotal values to be extracted in order to guide the designer in deriving a reliable and tight WCET.

References

[1] *OTAWA*. Available at: http://www.irit.fr/recherches/ARCHI/MARCH/OTAWA/doku.php?id=doc:computing_a_wcet.

[2] Ermedahl, A., Engblom, J., "Execution Time Analysis for Embedded Real-Time Systems," eds. Joseph, Y-T., Leung, S. H., Son, I. L., Chapman and Hall/CRC – Taylor and Francis Group, 2007.

[3] Lokuciejewski, P., Marwedel, P., *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems – Summary and Future Work* (Springer: Netherlands), pp. 229–234, 2011.

[4] *AbsInt GmbH*. Available at: http://www.absint.com/ait/analysis.htm.

[5] *Tidorum Ltd*. Available at: http://www.bound-t.com/.

[6] *NUS*. Available at: http://www.comp.nus.edu.sg/∼rpembed/chronos/.

[7] *IRISA*. Available at: http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29&Itemid=&lang=fr.

[8] *MRTC*. Available at: http://www.mrtc.mdh.se/projects/wcet/sweet/DocBook/out/webhelp/index_frames.html.

[9] Kirner, R., Puschner, P., Wenzel, I., "Measurement-based worst-case execution time analysis using automatic test-data generation." *4th Euromicro International Workshop on WCET Analysis*, pp. 67–70, 2004.

[10] *Rapita Systems Ltd*. Available at: http://www.rapitasystems.com/products/rapitime/how-does-rapitime-work.

[11] Carnevali, L., Melani, A., Santinelli, L., Lipari, G., "Probabilistic Deadline Miss Analysis of Real-Time Systems Using Regenerative Transient Analysis." In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, Versaille, pp. 299–308, 2014.

[12] Santinelli, L., Morio, J., Dufour, G., Jacquemart, D., "On the Sustainability of the Extreme Value Theory for WCET Estimation." 14th International Workshop on Worst-Case Execution Time Analysis, Versailles, pp. 21–30, 2014.

[13] *Proartis: Probabilistically Analysable Real-Time Systems*. Available at: http://www.proartis-project.eu/.

[14] *Probabilistic real-time control of mixed-criticality multicore and many-core systems (PROXIMA)*. Available at: http://www.proxima-project.eu/.

[15] *Rapita Systems Ltd*. Available at: https://www.rapitasystems.com/products/rapicover.

[16] *The International Electrotechnical Commission. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7, 2nd Edition, Requirement C.5.20 (Performance Modeling)*, Geneva, p. 99, 2010. IEC 61508.

[17] Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B., "The Mälardalen WCET benchmarks – past, present and future." In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)* Brussels, Belgium, pp. 137–147, 2010.

[18] Onera. *Onera – DiagXTrm*, Available at: https://forge.onera.fr/projects/diagxtrm2.

[19] MTime, *Vienna real-time systems group*, Available at: http://www.vmars.tuwientuwien.ac.at.