CARL VON OSSIETZKY universität OLDENBURG

Carl von Ossietzky Universität Oldenburg

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Legacy Software Migration based on Timing Contract aware Real-Time Execution Environments

Von der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Frau Irune Yarza Perez
geboren am 7. Juni 1992 in Galdakao (Spanien)

Irune Yarza Perez: *Legacy Software Migration based on Timing Contract aware Real-Time Execution Environments*

Gutachter:
Prof. Dr.-Ing. Wolfgang Nebel

Weitere Gutachter:
Prof. Dr.-Ing. Axel Hahn

Tag der Disputation:
26. Juni 2020

Abstract

The fast evolution of embedded systems market is generating interest on improved embedded microprocessor technologies. As a consequence, the obsolescence period for the underlying hardware is being shortened. As this happens, software designed for those platforms, that might be functionally correct and validated code, may be lost in the architecture and peripheral change. As embedded systems usually have real-time computing constraints, the legacy code retargeting issue directly affects real-time systems. When it comes to legacy code migration, binary translation appears to be a standard approach. However, when dealing with real-time legacy code, not just the functional behaviour, but also the timing behaviour has to be preserved. In the direction to solve this problem, the overall goal of this research line is to enhance the latest low-overhead machine-adaptable binary translation tool with the ability to preserve the timing behaviour on the translated binary. Through a feasibility study, a static binary translation tool is selected, which is latter enhanced with a timing enforcement mechanism that at the same time provides means for validating the enforced timing behaviour on the new platform using formal timing specification in the form of contracts.

Keywords: real-time, legacy software, migration, time contract

Kurzzusammenfassung

Die schnell fortschreitende Marktentwicklung für eingebettete Systeme weckt das stetige Interesse an verbesserten eingebetteten Mikroprozessoren und den zugehörigen Hardwareplattformen (sog. Systems-on-a-Chip). Daraus ergibt sich als unmittelbare Folge, dass auch die Veraltung von existierenden eingebetteten Mikroprozessoren und der zugrundeliegenden Hardware immer schneller voranschreiten. Eingebettete Software die einmal geschrieben wurde und für den jeweils eingesetzten Prozessor und die umgebenden Hardwareplattform hin angepasst, optimiert und validiert wurde, lässt sich in der Regel nicht auf triviale Weise auf eine neue Prozessorplattform portieren. Dies wird noch einmal dadurch erschwert, dass eingebettet System neben den funktionalen Eigenschaften auch noch Echtzeiteigenschaften einzuhalten haben. Bei einer Portierung der alten Software auf eine neue Hardwareplattform müssen demnach die Einhaltung der vollen Funktionalität und der zeitlichen Eigenschaften garantiert werden. Traditionelle Ansätze zur Portierung alter und hardwarearchitekturabhängiger Software auf neue Prozessorsystem nutzen häufig sog. „Binary Translation“ Techniken. Diese Techniken können in der Regel nur die Erhaltung der korrekten Funktionalität nicht aber die zusätzliche Einhaltung der Zeiteigenschaften sicherstellen. Im Rahmen dieser Arbeit wird die Verwendung der „Binary Translation“ Technik in Kombination mit Zeit- und Kontrollblöcken zur automatisierbaren funktionalitäts- und zeiteigenschaftserhaltende Übersetzung von bestehender echtzeitkritischer Software auf neue Hardwarearchitekturen untersucht. Im Rahmen der Arbeit wurde zunächst eine Vergleichs- und Realisierbarkeitsstudie verschiedener „Binary Translation“ Techniken und Werkzeuge durchgeführt. Das in diese Studie ausgewählte Werkzeug wurde mit kontraktbasierten Zeitspezifikations- und Zeitkontrollblöcken angereichert, um die korrekte Funktionalität und eine Einhaltung des spezifizierten und akzeptablen Zeitverhaltens auf der neuen Hardwareplattform zu ermöglichen.

Schlagworte: Echtzeit, veraltete Software, Portierung, Zeitvertrag

Acknowledgement

I would like to express my deep and sincere gratitude to everyone who gave me suggestion, guidance and support over the past few years.

I would like to express my special thanks to my supervisor, Prof. Wolfgang Nebel, as well as to Dr. Kim Grüttner and Dr. Mikel Azkarate-askatsua who gave me the golden opportunity to carry out this project in collaboration between the *Dependable Embedded System* group of Ikerlan Research Centre (Basque Country, Spain), and the *Hardware/Software Design Methodology* group of OFFIS (Oldenburg, Germany). I would also like to thank them for their continued support, encouragement and wise advise that guided me towards the goals of this thesis.

I am grateful to all of those with whom I have had the pleasure to work during this time. Thanks to the Ikerlan family, specially to those members who I call friends: Irune, Iñaki, Imanol, Ane, Charly, Itxaso, Asier, Peio, Aritz, Ángel and Blanca. Many thanks also to my colleagues at OFFIS, specially to Philipp, for their hospitality and support during my stays in Oldenburg, you made me feel at home.

I am lucky to have a family and friends that always supported me on whatever decision I made. Thanks to my closest friends, that during these years where by my side: Jenny, June, Irati, Nekane, Oihane, Francisca, Lukas, and Lena.

Most of all, I am forever grateful to my family, specially my parents Eva and Iñigo, my brother Gorka, and Rossana for their comprehension, motivation and for all the opportunities they gave me. And to my boyfriend Bingen, for his sincere love and optimism even in the difficult moments, this thesis would not have been possible without his endless support.

Contents

1 Introduction

Due to the great expansion and fast evolution of embedded systems' market, companies within the embedded system industry are facing a relentless demand for increasingly stringent requirements such as better performance, increased dependability, and energy efficiency, while offering a cost-effective product within a reduced time-to-market. This transition to next generation embedded systems is being encouraged by the rapid development of computing architectures. As a consequence, the obsolescence period of embedded systems is being shortened and maintenance is gaining importance in software life-cycle. Several empirical surveys analysed the percentage of software maintenance in software life-cycle over the years [71, 24, 81, 70, 79, 76, 37, 92]. These studies point out that the lower bound of maintenance cost continues to increase over the years [93]. In fact, most projects are no longer developed from scratch and need to deal with legacy systems and their integration.

Legacy systems are characterized by some particular properties:

- They usually run on obsolete hardware and operating system which are slow and expensive to maintain [97].
- Use customized and deprecated toolchain(s) and the compilation process takes a very long time [93].
- Have no or outdated documentation and original developers or users are no longer available [93].
- They are essential for the company [17] since they comprise business knowledge [94].

Due to their nature and particular properties, legacy systems present a complex scenario in software maintenance and evolution. The process of updating legacy systems is usually complex, error-prone, time-consuming and requires high cost investment.

Regardless of the strategy chosen to cope with legacy systems, modernization or complete replacement, both comprise the costly and complex reengineering process [1]. Retargeting, which consists on the analysis and migration of software to a new hardware platform meeting new requirements and easing future maintenance is one of the existing forms of reengineering. Existing retargeting methods can be classified into the following three categories:

- **New development:** which implies re-writing the legacy application. This approach is commonly associated with huge cost and many risks;
- **Wrapping or a black box approach:** which enables legacy software components to be reused without any knowledge of their implementation. However, this approach might only be a short term solution, since basic problems are not necessarily solved; and
- **Migration or white box approach:** which analyses existing legacy code and extracts the relevant parts to be reused. This is an intermediate between new development and wrapping, it is more complex than wrapping but less risky than new development.

Based on the retargeting methods, research efforts have provided several solutions, whereof the most common approaches can be listed as follows [27]:

- **Retargeting compilers to a new architecture:** a specially-constructed compiler is used to translate the system and its applications into the machine language for the new architecture. This approach is commonly used when the difference between source and target architectures is not very great or when legacy source code is at sufficiently high level that is effectively architecture independent. This approach requires that the legacy source code is available, which is not always the case in legacy systems.
- **Interpretation:** the Instruction Set Architecture (ISA) of the old architecture is emulated, by software interpreter, on top of the new architecture. This approach generally results in performance degradation.
- **Binary translation:** existing binaries (for the legacy ISA) are translated to binaries for the new architecture, either statically (before runtime) or dynam-

[1] Reengineering was described by Tilley and Smith [90] as "*[...] the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer*".

ically (during execution) [2]. This approach could results in the binary code growing significantly; as a consequence performance could be reduced.

Nonetheless, Binary Translation (BT) appears to be a standard approach, as it can be applied when there is a great difference among source and target architectures and either to port application- or system-level legacy code. Moreover, it can provide greater performance rates than interpretation, since translated code is reused avoiding retranslation.

1.1 Scope

Besides being the default approach for code retargeting, BT techniques have been implemented for many other purposes. Some of the utilities include fast emulation and simulation of instruction sets, binary instrumentation, analysis and optimization, resource protection and management (safety) and software security enforcement.

However, while BT techniques offer great advantages, there are challenges that have limited their adoption in the embedded systems domain. Embedded systems usually have tight memory and stringent performance requirements, whereas BT has been traditionally employed on computer systems with large memory and high performance capacities. Moreover, many of the embedded computing systems in today's technology market have strict timing and certification requirements, Critical Embedded Real-Time Systems (CERTS) are such an example, but BT techniques have not yet successfully came through this field, since satisfying the strict requirements in the CERTS domain is challenging itself [5].

Although binary translation has been successfully applied for legacy software migration, it is necessary to consider that Real-Time (RT) embedded systems usually run bare-metal code. Therefore, the binary has dependencies beyond the functional code, such as libraries and legacy hardware dependent firmware and/or drivers. When it comes to port legacy code hosted by an Operating System (OS) based platform, dependencies are commonly solved by porting user-level code and reusing the Application Programming Interface (API) to the underlying OS. UQBT [29], Crossbit [98], and LLBT [86] are some of the binary translation solutions that port user-level code maintaining the underlying OS. Moreover, when dealing with RT legacy code migration, not just the functional properties, but also the timing behaviour has to be preserved. To the authors knowledge, Cogswell and Segall [33] and Heinz [48]

[2] Dynamic binary translation defers from interpretation in that dynamic binary translation saves generated target machine code to be reused in future runs.

are the only ones who considered timing on their proposed retargeting solutions. However, none of them presents a portable solution, therefore, industry still needs an embedded RT legacy software retargeting solution that can be easily ported to different source and target architectures.

In the direction to approach an embedded real-time legacy software migration solution, this thesis will set the focus on the timing issue, therefore, the overall goal is to provide a migration path to real-time legacy software by integrating a portable timing enforcement mechanism into a machine-adaptable binary translation tool. The proposed solution should also provide means to validate the enforced timing behaviour on the new platform. Figure 1.1 depicts the scope of this research work.

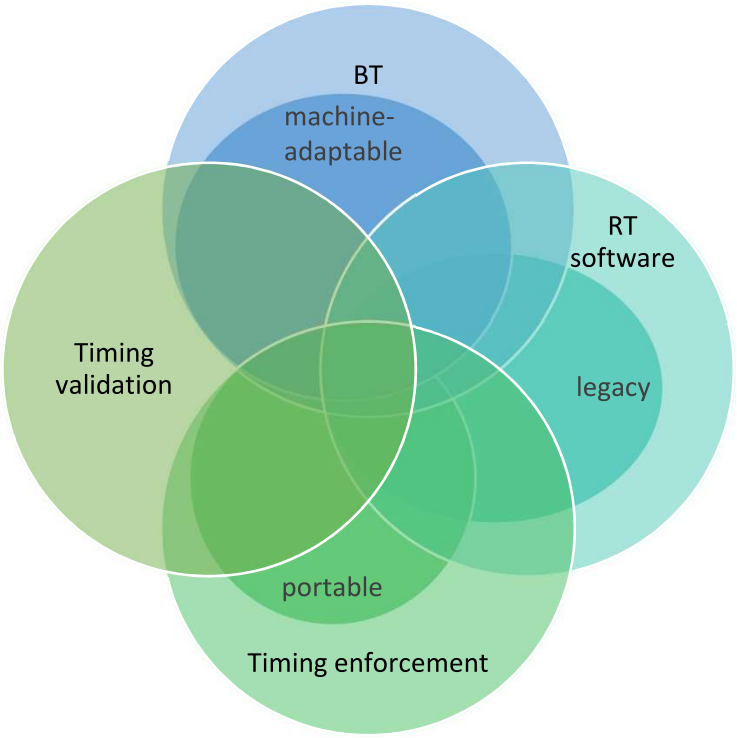


Figure 1.1.: Scope of this thesis.

1.2 Research Questions and Methodology

On the way to achieve the defined goal, this thesis intends to answer to the following research questions:

RQ1 How can legacy code be ported to a new architecture while maintaining its functional and timing behaviour?

RQ2 Under which constraints is it feasible to port RT legacy code using BT techniques?

RQ3 How can expert knowledge related to legacy timing be expressed and annotated to the legacy application?

RQ4 How can the annotated timing behaviour be preserved during the migration process?

RQ5 How can the timing behaviour on the new architecture be verified?

To determine existing techniques to port real-time legacy software, the literature is reviewed first, and two open-source machine-adaptable binary translation tools, one dynamic and the other static, are selected. As a first step, this thesis analyses and compares the suitability of the selected binary translation tools for their use in a real-time property conserving migration process. Through this feasibility study, the most suitable translation tool is selected, which is latter enhanced with a timing enforcement mechanism that at the same time provides means for validating the enforced timing behaviour on the new platform using formal timing specification in the form of contracts. The proposed real-time legacy software migration path is finally evaluated using three use-case applications.

1.3 Thesis Organization

This thesis is organized in eight chapters. The background and basic concepts on which the work described in this thesis is based are introduced in Chapter 2. Once the basis are set, Chapter 3 discusses related scientific work covering timing-aware recompilation and binary translation. Then, Chapter 4 defines the contributions that cover, at some point, the existing gap. Next, in Chapter 5, the real-time legacy software migration approach is presented, which starts with the legacy system model definition. Once the solution is constraint to a specific application model, the lifting of legacy timing properties is described, followed by the process of testing timing as well as functional properties and reallocating time control blocks. The chapter concludes with the timing equivalent legacy software translation. The proposed migration path is then implemented as described in Chapter 6 and evaluated in Chapter 7. The evaluation discusses the results obtained in the feasibility study, block-level timing enforcement assessment and timing-aware static legacy software migration assessment through the example, industrial and multirotor applications. Chapter 8 concludes the thesis and presents an outlook on future research work in this area. Finally, Appendix A presents the algorithms for the systematic annotation

of legacy code using time measurement and control blocks, as well as the systematic transformation of annotated legacy code into formal timing specification for the latter timing validation.

2 Background and Basic Concepts

This chapter presents the background on which this thesis is based and explains basic concepts in the field of embedded systems (with a refinement to real-time embedded systems), timing specification, and legacy software migration.

2.1 Embedded Systems

Embedded systems have a great presence in our everyday life, many of the devices in common use today are controlled by embedded systems. We are surrounded by large systems enhanced with an embedded computing system that turns the overall system into an intelligent product. Hence, an embedded system is a controller with a dedicated function within a larger system, which has limited resources and often also real-time constraints. When it comes to embedded computer systems design, defining and understanding its architecture and the components that make it up is essential for a good design.

2.1.1 Architecture

Noergaard [75] presented the Embedded Systems Model shown in Figure 2.1. This model introduces in a modular representation the main elements present in embedded computer systems architecture. The *application software layer* is the software that defines systems functionality and implements most of the man-machine interface. The *system software layer* contains any software that supports the application, be it middleware, OS or device driver. The *middleware* acts as an abstraction layer between application and the underlying systems software component, either OS or device driver. The *OS* is a set of software libraries that provides an abstraction layer to middleware and application and manages multiple system software and hardware resources to ensure efficient and reliable system operation. Whereas *Device drivers* are the software libraries that directly interface with and control hardware. The *hardware layer* integrates most of the physical components in the embedded board

(i.e., processor, memory, Input/Output (I/O) devices and buses), the architecture layer is present in every embedded system. Instead, system and application software layers, which make up all the software in the computer system, may or may not be present in an embedded system.

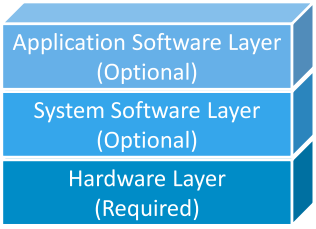


Figure 2.1.: Embedded Systems Model [75]).

2.1.2 Software

The components in an embedded systems hardware layer can only transmit, store and execute *machine code*, a hardware dependent binary language. Machine code, considered the fist generation in programming languages, was originally used to program computer systems. However, to make programming more efficient, programming languages evolved towards higher levels of abstraction. The second generation in programming languages was *assembly*, which consists on a set of hardware specific instructions that correspond to one or multiple machine code operations. Unlike assembly language, which is hardware-dependent and therefore considered as a *low-level language*, next generation languages are referred to as *high-level languages* since they are more independent (or even completely independent) of the underlying hardware layer. Languages such as C and Pascal with more English like expressions (third generation) or C++ and Java object-oriented languages (forth generation) are part of the high-level and very high-level programming language groups respectively.

2.1.3 From High-Level to Machine Code

Except for machine code, software written in any other programming language needs some kind of mechanism to generate equivalent machine code in order to be executed on top of the hardware layer. This mechanism is based on one or a combination of the following components: *compiler* and *interpreter*. The former translates source code into a particular target language at one time, whereas the later generates machine code interpreting source code line by line.

Compilers usually run on the host machine (typically non-embedded) and generate code for a different hardware platform, target machine, this type of compilers are known as *cross-compilers*. Moreover, the supported source/target code can vary among compilers. In the case of assembly language compilers (a.k.a. *assembler*), the generated code is always target machine code. In contrast, high-level programming language compilers can either generate machine code, other high-level code or assembly code. When the generated code is another high-level language code, this needs to be further compiled or interpreted to generated machine code. Whereas, when the compiler generates assembly code, it needs to run through an assembler.

Programming projects may contain multiple source code files compiled with independent compilation units. Moreover, routines commonly used in the application program are collected into *libraries* to ease their use in multiple projects, increasing the productivity of software developers. Libraries can be linked to the program either statically (before execution) or dynamically (during execution). Therefore, the *object files* resulting from the compilation process are then combined with any other required system libraries into a single binary program using a *linker*. The *executable file* resulting from the linking process contains code and statically linked data in binary representation. The executable file is stored in the embedded platform and the loader is in charge of reading the file and copying code and data into memory for its execution. During execution, a *dynamic linker* does the binding of symbols in *shared libraries*.

2.1.4 Abstraction Layers

Embedded systems are organized into several abstraction layers as shown in Figure 2.2 [54]. At the bottom layer, the *ISA*, which defines the instruction set for the processor, registers, memory and interrupt management, provides the abstraction between software and hardware layers. The ISA is divided into user ISA, visible to applications, and system ISA, visible to supervisor software. At the next level, the *Hardware Abstraction Layer (HAL)* allows that the OS interacts with the hardware at an abstract level rather than at a detailed hardware level. The *Application Binary Interface (ABI)* separates the OS and device drivers from middleware and application software. So, once the application has been compiled and linked, the ABI provides the interface between the executable file and the layer bellow. At the highest level of abstraction, the *API* mediates between the application program and underlying middleware and OS.

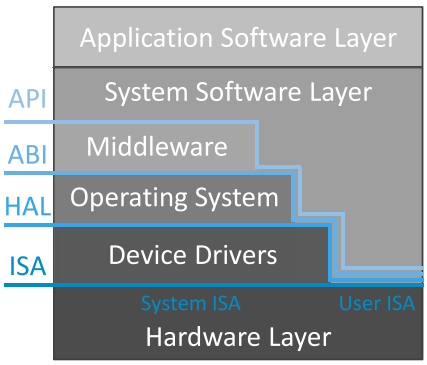


Figure 2.2.: Abstraction Layers and Embedded Systems Model.

Abstraction layers provide compatibility among systems. The ISA abstraction layer allows software built to a given ISA to run on any hardware that supports that ISA. The HAL enables the OS to be easily portable across different hardware. The ABI provides portability of applications and middleware across OSs, since binaries compiled to a specific ABI can run unchanged on a system with the same OS, device drivers and ISA. Whereas software using a given API can be ported to other platforms through recompilation.

2.2 Real-Time Embedded Systems

RT embedded systems make up the most important market segment for RT technology and computer industry in general [61]. As RT systems' state changes as a function of physical time, correctness of the entire system's behaviour depends not only on the correct logical results of computations, but also on the instant when these results are produced and shared with other subsystems.

The instant when the result must be produced is named a *deadline*. If the result is useful even after a *deadline miss*, the deadline is said to be *soft*, whereas otherwise it is a *firm* deadline. Moreover, according to the consequences of a deadline miss, a firm deadline is referred to as *hard* when a miss could lead to catastrophic consequences. According to the deadlines they must meet, real-time systems are classified into hard or soft. A real-time computer system is said to be hard or *safety-critical* when it must meat at least one hard deadline. In contrast, when no hard deadline exists, the computer system is classified as soft real-time.

2.2.1 Time Model

Given its close relationship, incorporating the notion of time into RT embedded systems is a key requirement [78]. RT is usually represented with a *timeline*, showing the progression of time from *past* into the *future*. The timeline consists of an infinite set of instants T, where an *instant* corresponds to a cut on this line. Any relevant happening at a particular instant is called an *event*, e. The event that decouples past from future and determines the present point in time is named *now*. The *value* of an event, denoted by $v(e_i)$, is the value of time at the instant when the event happened. An interval between two events, denoted by $d(e_i, e_j)$, is called a *duration*, an event itself does not take any time [58]. Figure 2.3 depicts the described time model [62].

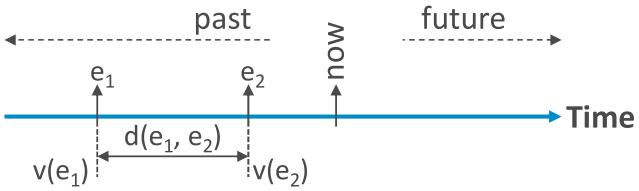


Figure 2.3.: Time model: timeline, events, and duration.

2.2.2 Real-Time Control Systems

RT control is a relevant application field in RT embedded system. Industrial plant automation was the first field for the application of RT control systems. However, many others followed and RT control systems are now widespread in consumer, industrial, medical, and military applications.

Embedded RT control systems are commonly decomposed into a set of self-contained subsystems: the *physical environment* to be controlled, the *real-time computer system* that controls this environment through the instrumentation interface, and, most often, a *man-machine interface* to interact with the system. If the RT computer system is *distributed*, computing resources will be distributed over several hardware resources, possibly at different locations, and connected to other (distributed) computer system.

Functional and Temporal Requirements

In a RT computer system, the systems *functional requirements* determine the functions that the real-time computer system must perform. With regard to RT control

system, functional requirements would consist of e.g., monitoring of significant state variables, detection of abnormal system conditions through alarm monitoring, inform and assist the operator in controlling the environment object, and execution of the control algorithm to drive the state variable to the set point. However, as already stated, the main characteristic of embedded real-time systems is the existence of temporal as well as functional requirements. The temporal requirements specify timing constraints for sequences of events. To understand where do this *temporal requirements* come from in a real-time control system, some basis of *control engineering* are needed.

Control Engineering

Real-time control systems commonly exhibit a highly regular pattern, consisting on a cyclic sequence of actions. Figure 2.4 shows representative events and intervals of a real-time control loop. To this end, a *cyclic representation of time* has been used [61], which better fits the cyclic pattern or a control system and human cognitive nature of time.

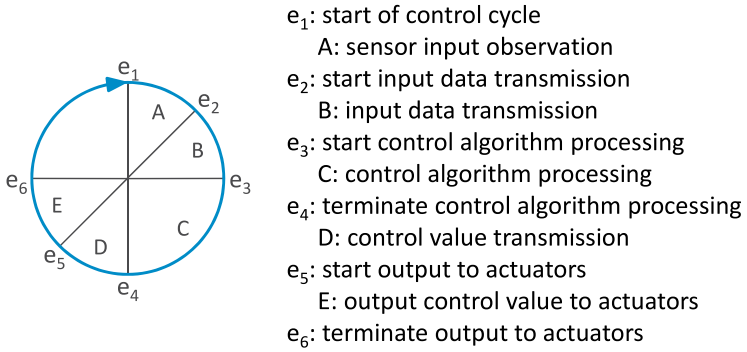


Figure 2.4.: Real-Time Control Loop – cyclic time representation.

The first task consists on collecting data by observation of significant state variables. However, because the state of the physical environment under control changes as a function of real time, an observation will only be accurate for a limited time interval. Therefore the computer system must sample state variables periodically. The constant duration between two sampling points ($d_{sampling}$) is called *sampling period* and is determined by the dynamics of the controlled system. In Figure 2.4 the sampling period corresponds to the time interval between consecutive e_1 events, a full control loop. Obtained input data is then transmitted and the control algorithm is executed next, which based on the error calculated from comparing the state variable with the corresponding set point value, computes the new value for the control variable. The last step consists on the output of the new control variable to

the actuator. The duration between the events associated with control cycle start and finish ($d_{computer} = d(e_1, e_6)$) is called *computer delay*, which includes apart from the time needed to perform the control computations, the time needed for I/O communication. The difference among the maximum and minimum computer delay is called computer delay jitter, $\Delta d_{computer}$. For a correct behaviour of the real-time system, the computer delay should be less than the sampling period. Moreover, the *computer delay jitter* should be a small fraction of the computer delay. The control algorithm can be designed to compensate a constant computer delay, but the uncertainty introduced by the delay jitter has an adverse effect on the quality of control.

2.3 Timing Property Specification

When it comes to RT embedded systems design, there is a need for means to specify timing properties along the design process in order to enable coherent reasoning about timing within complex scenarios. To this end, the MULTIC project [14] defined a *timing specification language*, which was then updated on [15]. The defined time specification language is based on *specification patterns*, which are natural language like statements. The syntax and semantics of specification patterns are defined in terms of timed traces that satisfy the pattern. *Timed traces* are based on a notion of sampled signals and determine the value of variables in the time domain.

A signal can describe continuous as well as discrete systems behaviour, as depicted in Figure 2.5. If a signal has non-absent values only for $t \in T \subset \mathbb{T}$, where T is some discrete set, is said to be a *discrete-event signal*. Whereas if a signal is non-absent for all time points, two further signal types are distinguished: *discrete-evolution signals*, when value changes happen only at time points, or *continuous-evolution signals*. The MULTIC Time Specification Language (MTSL) focusses on discrete-event signal based systems.

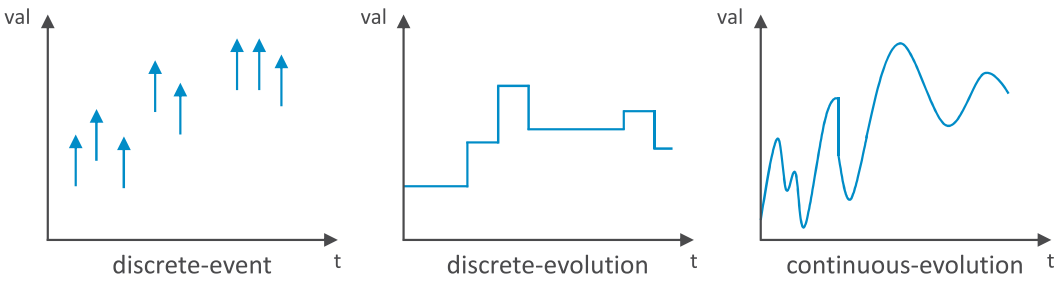


Figure 2.5.: Signal types.

Signals are only visible at ports and every behaviour observable at a given port p is restricted its value domain, denoted by Σ_p. Therefore, semantics of port behaviour can be represented in terms of timed traces:

Definition 2.3.1. (Timed Trace). A timed trace observed at port p, is defined as a tuple $\omega_p = (t_i, \sigma_i)_{i\in\mathbb{N}}$, where $(t_i)_{i\in\mathbb{N}}$ is an infinite sequence of monotonic time instances and, for each time instance, $\sigma_i \in \Sigma_p$ denotes the corresponding element from the value domain of port p. Timed traces are required to be non-zero, therefore, for each $t \in \mathbb{T}$ exists a timed trace (t_i, σ_i) such that $t_i \geq t$. A set of timed traces observed at port p is denoted by $\Omega_p = \{\omega = (t_i, \sigma_i)_{i\in\mathbb{N}}\}$.

The same way, a set of timed traces observed at a port set $P = \{(p_i)_{i\in\mathbb{N}}\}$ is denoted by $\Omega_P = \{\omega_P = (t_i, \vec{\sigma_i})_{i\in\mathbb{N}}\}$, where $\vec{\sigma_i} = (\sigma_1, \ldots, \sigma_n) \in \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$.

Projection $\omega_P|_q$ of traces over a port set P to port $q \in P$, where $\omega_P|_q = (t_i, \sigma_i^q)_{i\in\mathbb{N}}$ if and only if $\omega_P = (t_i, (\ldots, \sigma_i^q, \ldots)_i)_{i\in\mathbb{N}}$.

2.3.1 MULTIC Time Specification Language

MTSL is based on timing specification patterns (defined in terms of Backus-Naur Form (BNF) grammar), which consist of natural language like statements composed of fixed keywords and parameters specified by the user.

Within the MULTIC project, *parameters* are written in *slanted* font, whereas **keywords** are written in **bold** font and additionally enclosed in quotation marks (as '**keyword**') when they are hardly recognisable. Optional parts are enclosed in brackets and followed by a question mark, such as [optional part]?. Whereas parts that may occur zero or more times are also enclosed in brackets but followed by a star, as for example [repeated part]*. Grammar patterns are composed (from left to right) of a name separated with a double colon :: from the definition. Alternatives within the definition are separated by a vertical bar denoting for this | that.

As stated above, timing specifications describe relations among events, which are only observable at ports and fixed to a corresponding value domain. A timing specification refers to one or multiple events, however, the event value may or may not be of importance. Therefore, event specifications are specified as follows:

EventSpec :: *Port* | *Port* '.' *EventValue*

The *EventValue* parameter is deliberately left open, it could consist of labels as well as (complex) values. If an *EventValue* is not specified in the *EventSpec*, any event observed at the *Port* is considered.

In fact, given a timed trace $\omega = (t_i, \sigma_i)_{i \in \mathbb{N}}$ and an event $(t_i, \sigma_i) \in \omega$, it is said to satisfy the event specification, denoted as $(t_i, \sigma_i) \models$*EventSpec*, if either *EventSpec* specifies a port and σ_i corresponds to its value domain or *EventSpec* specifies an event value and σ_i equals to it.

Timing specifications may refer to a single event, a sequence or a set of events:

EventExpr :: *EventSpec* | '**(**' *EventList* '**)**' | '**{**' *EventList* '**}**'
EventList :: *EventSpec* ['**,**' *EventSpec*]*

Extending the notion of satisfaction to event expressions, a timed trace $(t_i, \sigma_i), \ldots, (t_{i+n-1}, \sigma_{i+n-1})$ is said to satisfy an event sequence $es = (e_1, \ldots, e_n)$ if every event $(t_{i+k-1}, \sigma_{i+k-1})$, $1 \leq k \leq n$, satisfies the event specification e_k. While $(t_i, \sigma_i), \ldots, (t_{i+n-1}, \sigma_{i+n-1})$ satisfies an event set $es = \{e_1, \ldots, e_n\}$ if a sequence $(e_{s_1}, \ldots, e_{s_n})$ exists that satisfies $\{e_{s_1}, \ldots, e_{s_n}\} = \{e_1, \ldots, e_n\}$.

Timing specifications can either refer to a time point or an interval:

TimeExpr :: *Value Unit*
Boundary :: '**[**' | '**]**'
Interval :: *TimeExpr* | *Boundary Value* '**,**' *Value Boundary Unit*

Time units and values in time expressions are restrict to usual time units and simple numbers:

Unit :: **s** | **ms** | **us** | **ns**
Number :: **0** .. **9**[**0** .. **9**]*
Value :: *Number* | *Number* '**.**' *Number*

In the following, some of the patterns defined within the MULTIC Timing Specification Language (those relevant for this work) are presented. Detailed information on the Timing Specification Language can be found in [15], Chapter 3.

Event Occurrence

To describe a repetitive event occurrence in a particular port, the *Repetition* pattern is introduced. This pattern complies with the usual meaning of periodic patterns, as well as patterns with minimum and maximum inter-arrival times. The *Repetition* pattern is defined as follows:

Repetition :: *EventList* **occurs every** $Interval_1$ [**with** *RepetitionOptions*]?.
RepetitionOptions :: *Jitter* [**and** *Offset*]? | *Offset* [**and** *Jitter*]?
Jitter :: **jitter** *TimeExpr*
Offset :: **offset** $Interval_2$

Within the *Repetition* pattern, parameter $Interval_1$ determines the minimum and maximum time period between subsequent occurrences of the *EventList*. The *Jitter* defines an additional delay between subsequent occurrences of the *EventList*, while the *Offset* defines a delay interval for the first occurrence of the *EventList*.

Definition 2.3.2. (Repetition pattern semantics). Semantics of the repetition pattern "EL **occurs every** I **with jitter** J **and offset** O." is defined as the set of timed traces $(t_i, \vec{\sigma_i})_{i\in\mathbb{N}}$ such that $\vec{\sigma_i}$ corresponds to the event list EL, and $t_i = u_i + j_i \wedge u_0 \in O \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$, where $I = (P^-, P^+)$ is the specified interval (where (and/or) may be replaced, to indicate a closed upper and/or lower bound, by [and/or] respectively), $O = [O^-, O^+]$ is the offset interval, and $J \geq 0$ is the jitter. $P^- > 0$ is required.

For large jitter (i.e., $J > P^-$), the language definition is not quite correct because t_i may not be monotonic ordered any more. This issue is corrected by reordering the trace through a bijective function $k : \mathbb{N} \to \mathbb{N}$, where $(t_{k(i)}, \sigma_{k(i)})_{i\in\mathbb{N}}$ defines a trace so that $(t_{k(i)})_{i\in\mathbb{N}}$ froms a monotonic sequence again. Such trace is part of the pattern for which holds $t_{k(i)} = u_i + j_i \wedge u_0 \in O \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$.

Figure 2.6 shows multiple pattern instances with different parameters. In the first pattern, a minimal instance of the pattern is shown, with no jitter and no offset. The second pattern adds an up to 5ms jitter. The light blue bars in the time-line mark the period intervals as for the first patter, showing that the jitter is "added" to the "baseline" periodic behaviour. The third pattern defines a period interval (between 20 and 25ms). As none of the first three patterns defines an offset, the first event always occurs at time point 0. On the contrary, the forth pattern defines an offset in the [0, 10]ms interval. Therefore, the first event occurs somewhere in the interval (for example at 5ms), whereas the time interval between two successive event occurrences is in the interval [20, 25]ms.

Reaction Constraints

To specify a forward delay over events, event sets and event sequences, the *Reaction* pattern is introduced:

Reaction :: **whenever** *EventExpr* **occurs then** *EventExpr* **occurs within** *Interval* [**once**]?.

Definition 2.3.3. (Reaction pattern semantics). Semantics of the reaction pattern "**whenever** es_1 **occurs then** es_2 **occurs within** I.", where es_1 and es_2 are either a set or a sequence of events that contain k and l events, respectively, is defined as

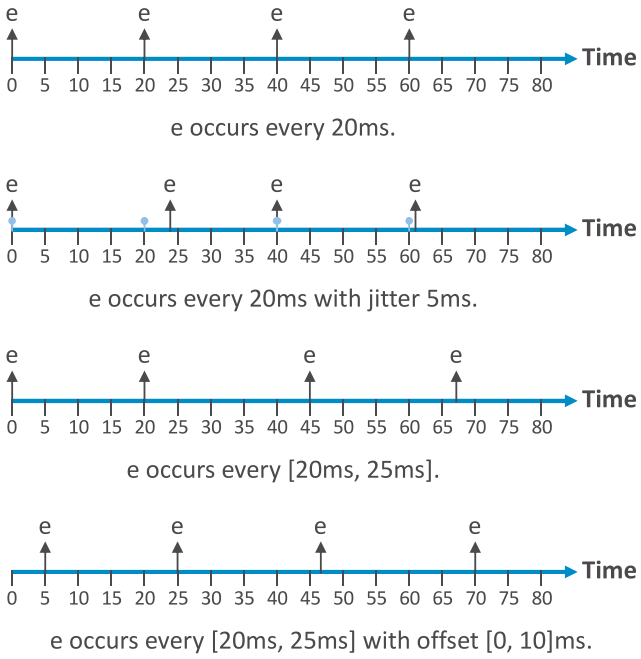


Figure 2.6.: Event occurrence pattern examples.

the set of timed traces $(t_i, \sigma_i)_{i \in \mathbb{N}}$ such that $\forall (t_i, \sigma_i) \dots (t_{i+k-1}, \sigma_{i+k-1}) \models es_1 : \exists\, j \geq i + k : (t_j, \sigma_j) \dots (t_{j+l-1}, \sigma_{j+l-1}) \models es_2 \wedge t_{j+l-1} - t_{i+k-1} \in I$.

The optional keyword **once** determines that the patter fails if more than one reaction occurs within the determined time window, meaning that only one $j \geq i + k$ exists such that the corresponding sequence satisfies es_2.

Figure 2.7 depicts multiple examples of the reaction pattern. The first time-line shows a fragment of a pattern instance where event f occurs within an interval of [15, 25]ms since event e. In contrast to the first pattern, the second pattern forbids multiple instances of event f within the specified interval due to the keyword **once**. The third pattern defines an event sequence (f, g) instead of a single event as the reaction to event e.

Causal Event Relations

Besides being able to reason about the timely behaviour of events, it is also important to be able to reason about relation of events. This can be captured by specifying the order of occurrence of different events. The MTSL allows the definition of basic functional relations by assigning event values. However, more complex functional relations are not (yet) supported.

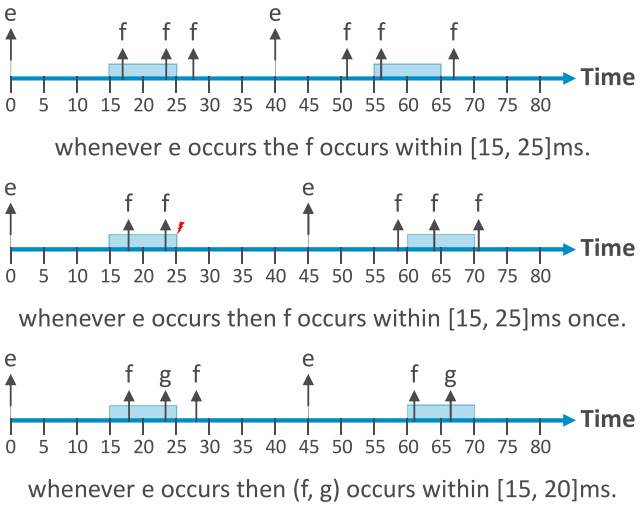


Figure 2.7.: Reaction pattern examples.

The formal definition of *causal event relations* is as follows:

Definition 2.3.4. (Causal Event Relation). Consider ports p_1 and p_2, and let Ω_{p_1,p_2} be the semantics of the ports. A causal event relation between p_1 and p_2 is a function

$$\triangleright(p_1, p_2) : (\mathbb{T} \times \Sigma_{p_1}) \rightarrow 2^{\mathbb{T} \times \Sigma_{p_2}}$$

where for all $\omega \in \Omega_{p_1,p_2}$ and for all event occurrences $(t_i, \sigma_i) \in \omega|_{p_1}$ exist (t_j, σ_j), …, $(t_k, \sigma_k) \in \omega|_{p_2}$ such that it holds $\triangleright(p_1, p_2)((t_i, \sigma_i)) = \{(t_j, \sigma_j), \ldots, (t_k, \sigma_k)\}$ and $t_i \leq t_j, \ldots, t_k$.

Moreover, causal event relations are transitive, which means that given three ports p_1, p_2, p_3 and causal event relations $\triangleright(p_1, p_2)$ and $\triangleright(p_2, p_3)$, then the causal event relation $\triangleright(p_1, p_3)$ is given by:

$$(t_j, \sigma_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge (t_k, \sigma_k) \in \triangleright(p_2, p_3)((t_j, \sigma_j)) \Rightarrow (t_k, \sigma_k) \in \triangleright(p_1, p_3)((t_i, \sigma_i))$$

Based on the causal event relation (see Definition 2.3.4), a causal version of the reaction pattern is defined:

CausalReaction :: **Reaction(***EventSpec* ',' *EventSpec* **within** *Interval.*

Definition 2.3.5. (Causal reaction pattern semantics). Semantics of the causal reaction pattern "**Reaction(**e_1, e_2**) within** I.", where e_1 and e_2 refer to p_1 and p_2,

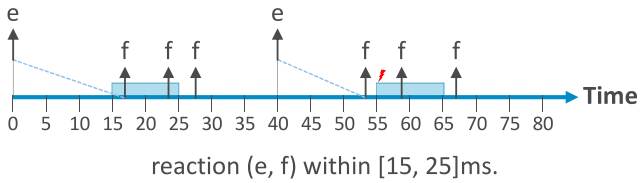


Figure 2.8.: Causal reaction pattern example.

respectively, is defined as the set of timed traces $\omega \in \Omega_{p_1,p_2}$ where for all $(t_i, \sigma_i)_{i\in\mathbb{N}} \in \omega|_{p_1}$, $(u_i, \rho_i)_{i\in\mathbb{N}} \in \omega|_{p_2}$, and for all event occurrences $(t_i, \sigma_i) \in (t_i, \sigma_i)_{i\in\mathbb{N}}$ such that $\sigma_i \models e_1$, holds $\triangleright(p_1, p_2)((t_i, \sigma_i)) \neq \emptyset$ and $((u_j, \rho_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge p_j \models e_2) \Rightarrow u_j - t_i \in I$.

Figure 2.8 shows an example of the causal reaction pattern. The dashed light blue line indicates that those events are related by the definition of causal reaction event relation. It is due to the causal relation, the causally related instance of event f does not occur within the defined time interval, that the pattern is violated within the second occurrence of event e, in contrast to the first example in the non-causal reaction pattern (see first example in Figure 2.7).

2.4 Legacy Migration Techniques

When it comes to run code from a given ISA in a different architecture, there are two main techniques available: Emulation and Simulation. Although both solutions provide means to run code compiled for a given ISA in a different hardware platform, the purpose of each technique is different. In the following the differences among emulation and simulation are presented first, followed by a description of binary translation as a technical solution to emulation.

2.4.1 Emulation vs. Simulation

Emulation and simulation terms are commonly used interchangeably, since both provide means to run software in platform different from that it was designed for, however there is a great difference among them.

On the one hand, emulation is the process of replicating the externally observable behaviour of a system with no regard for how the system functions internally. In fact, the internal state of the emulator does not have to accurately mimic the internal start

of the source that is being emulated on the target processor [1]. The main purpose of emulation is to be used as a substitute to the original device or system, so it must operate close to real-time.

On the other hand, the simulation process models the internal state of the source on the target processor. The main purposed of simulation is to analyse and study the source system. All in all, the simulator has to model the internal state of the source system with sufficient detail according to the purposed of the analysis. Unlike emulators, a simulator may run far slower than real-time.

Given that the purpose of the work presented in this thesis is to port a legacy application to a new hardware platform preserving its functional as well as timing requirements, binary translation is presented as a technical solution to emulation.

2.4.2 Binary Translation

The main purpose of BT techniques was architecture compatibility, in fact, BT is the default approach for legacy code migration. However, BT techniques have been implemented for many other purposes, such as fast simulation of instruction sets, e.g. Desoli et al. [35], Dehnert et al. [34], Bellard [16], and *Imperas Ltd. Open Virtual Platforms (OVP)* [56], for understanding the counterparts of different hardware designs and exploring novel ideas before the real hardware is available; binary instrumentation, e.g. Bruening et al. [20], Luk et al. [67], Hazelwood and Klauser [47], Nethercote and Seward [74], and Lyu et al. [68], where code is injected into a running process for debugging, simulation or profiling purposes; binary optimization, e.g. Bala et al. [13], Chen et al. [26], and Lu et al. [66], which aims to improve the performance of an executable program; and software security enforcement, e.g. Scott et al. [83], Kiriansky et al. [59], and Hu et al. [53].

BT is a technique that transforms existing binaries compiled for the *source ISA* (M_s) into binaries for the *target ISA* (M_t)[1]. In a binary translation process, the basic unit of translation is referred to as a *Basic-Block (BB)* and consists of a sequence of instructions that are likely to run as a whole, since the code block has just one entry and one exit point.

The first binary translators followed a *direct translation* approach, from source to target machine-code. However, the fact that BT is highly dependent on the source and target architectures hinders reusability. Therefore, researchers adopted the

[1] Some authors refer to source/target terms as target/host.

general approach in portable compilers, where machine-dependent and machine-independent concerns are separated through an *Intermediate Representation (IR)*, which acts as an abstraction layer between source and target ISAs. This approach eases the adaptation of the translation tool to multiple source and target architectures, since the NxM translation graph is transformed into an N+M graph (see Figure 2.9) [72]. Therefore, the resulting *IR based translation* process consists on the following three parts: the *front-end* which deals with the source machine-code and does the decoding processor; the *middle* that works at the IR-level and conducts the analysis and optimization phase; and the *back-end* that generates target machine-code through the encoder. The middle will be common for all source/target architectures, whereas a new front-end/back-end needs to be implemented to support a new source/target architecture respectively. A binary translation tool is said to be machine-adaptable when it can easily and inexpensively handle different source and/or target machines.

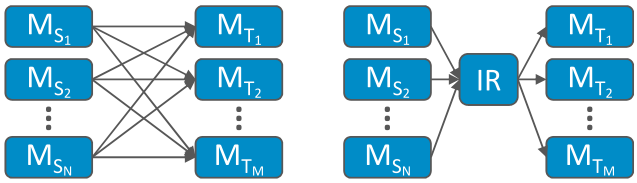


Figure 2.9.: Direct vs. IR-based BT. Direct BT = NxM binary translators (left side). IR-based BT = N front-ends + M back-ends (right side).

According to the supported input code, binary translators can be divided into two groups: user-level or system-level. If the tool supports the translation of a full software stack, including the legacy application, middleware and OS, it is known as *system-level translator*. Whereas a *user-level translator* support just the translation of the legacy application, implementing an interface at the ABI level or higher.

Moreover, according to the time when translation is performed, BT systems can be classified into two categories: Dynamic Binary Translation (DBT) and Static Binary Translation (SBT). *Dynamic Binary Translation* systems translate the binary code at run-time, during program execution, therefore this technique can easily handle indirect branches and can perform optimizations based on program's run-time behaviour. However, as translation and optimization time counts for a part of the execution time (which incurs execution overhead), dynamic translation approaches cannot perform aggressive optimizations. Whereas, *Static Binary Translation* systems translate the binary code offline, before the program is executed, so static translation approaches can perform whole program optimization without influencing run-time overhead, but they must deal with code discovery, code location and self modifying

code issues effectively to be considered a real solution [2]. Some researches, such as Shen et al. [85], have even developed hybrid binary translation techniques to take advantage of the strengths of both methods.

2.4.3 Static Binary Translation

SBT is composed of two separate phases: translation and execution phase. Figure 2.10 presents the structure of the translation phase in an IR-based SBT system.

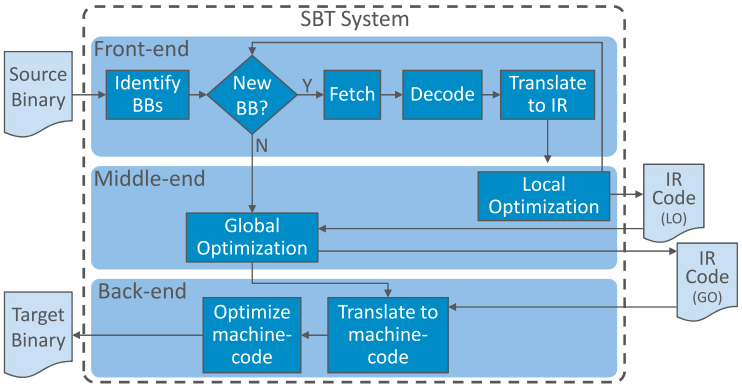


Figure 2.10.: Static Binary Translation flow diagram.

The front-end, which as already mentioned is a source machine dependent module, generates the intermediate code from the source machine code. To this end, the input binary is first loaded, then BBs are identified and for each BB machine code is disassembled/decoded to assembly code and translated into a machine-independent representation. At a machine and system independent level, the middle-end analyses the generated IR code for each BB and performs machine-independent optimizations. This type of optimization is known as *local optimization* as it is applied at a BB level. The translated and optimized BB is stored and once every BB has been translated, *global optimizations* are applied, which leverage relations among BBs. Finally, the back-end, which is a target machine dependent module, translates the optimized IR code on each BB into target machine code. The generated target machine code can be further optimized taking advantage of special characteristics on the target system to apply machine-dependent optimizations.

For the execution phase, some SBT systems have a fall-back mechanism used by the translator to effectively solve code location problems and to emulate access to hardware devices (in such a case that respective instructions have not been previously translated by the translator).

[2] Self modifying code is not a common practice.

2.4.4 Dynamic Binary Translation

In a DBT system both translation and execution are performed simultaneously, the legacy binary is dynamically analysed and translated into target machine code during runtime. In other words, program regions are translated on demand, if a new source code block needs to be translated the translator is invoked to generate the corresponding machine code using either a compiler or an emulator. Figure 2.11 shows the structure of an IR-based DBT system.

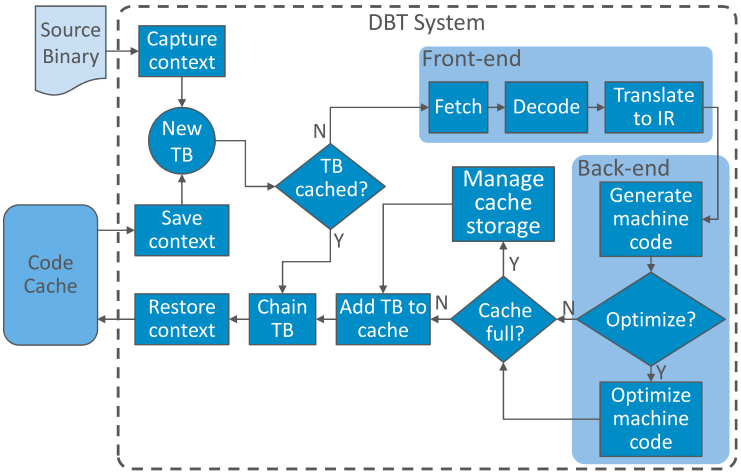


Figure 2.11.: Dynamic Binary Translation flow diagram.

The source machine dependent front-end generates intermediate code for a block of source code according to the address pointed by the source machine Program Counter (PC) value. The source code block usually corresponds to a BB, however, if the BB is too large the translator takes just a fragment of the BB. To generate the IR code, the block of code to be translated, Translation Block (TB), is fetched, decoded and then translated into an intermediate representation. Then the back-end translated the generated intermediate code into (unoptimized) target machine code and stores the resulting machine code in the code cache to be reused in future runs. When the cache is full, the cache manager is in charge of deciding whether to evict part or all the translated code, or to increase the cache size if possible. The simplest cache eviction policy (FLUSH) consist on erasing the entire content of the cache.

To reduce the overhead derived from context switched, code chaining is implemented on the DBT system, direct branches are replaces with a jump to the corresponding target code block, this way context switches are avoided. Moreover, as SBT systems do, DBT systems do also optimize translated code. However, as optimization counts on runtime, for dynamic binary optimization to be profitable a proper trace selection strategy is needed. The most common strategy is hot paths detection. The translator

keeps track on the number of times a block has been executed and if a threshold is reached, optimizations are applied to generate optimized target machine code. Some DBT tools do not translated the code until a threshold is reaches, so on the first runs the generated intermediate code is emulated and just frequently used code is translated and cached.

3 Related Work

Given that legacy software migration is a common issue in industry it has been widely studied during the last decades. Therefore, this chapter provides an overview of existing migration solutions and places in context the work presented in this thesis. The analysed solution space covers solutions for a timing-aware recompilation of the legacy source code, as well as techniques to directly port the legacy binary, although most of existing binary migration solutions are not timing aware.

3.1 Timing-aware Recompilation

Software recompilation is a well known solution when it comes to port legacy software to a new ISA. However, recompilation to be applicable on the legacy migration process, the legacy source code must be available and it must be written in sufficient high level that it is independent (or almost independent) from the legacy hardware platform. Moreover, when it comes to port real-time legacy software, apart from functional requirements, timing requirements must also be preserved on the migration process. In the following, existing timing-aware recompilation solutions are covered. Although solutions exist for Ada [96, 63] and Real-Time Java [21, 95], the focus is set on solutions for C programming language, which is the scope of this work as well as the most common programming language when it comes to the development of small-scale embedded system running on bare metal or lightweight RTOS based platforms [23].

3.1.1 LET-based Software on E-machine

Resmerita et al. [80] proposed a systematic approach to apply real-time programming to legacy embedded control systems composed of time triggered as well as event triggered tasks with different priority levels. Their solution is based on the Logical Execution Time (LET) model [60], a real-time programming abstraction introduces with the time-triggered programming language Giotto [50]. The LET determines the time interval from reading task input to writing task output, regardless

of the time duration (within the defined interval) that task's execution actually takes. Then, a dedicated middleware ensures task realization, achieving an increased system robustness and predictability. Therefore, the legacy source code is first analysed and transformed into LET-based software by replacing existing code and inserting new code for the realization of the LET. The timing specification language they used to implement the LET is Timing Definition Language (TDL) [89]. The transformed code is then compiled into LET-scheduling instructions named E-code and interpreted at run-time by an Embedded Machine (E-machine) which was presented by Henzinger and Kirsch [49]. The E-machine is a virtual machine that provides a real-time interaction among software and physical processes. The E code (executed by the E-machine) is platform-independent and supervises the environment time (interaction of software processes with physical processes). Whereas the E-machine interprets the E code taking care of platform time (time at which software processes execute on a specific platform, scheduling). There are several implementations of the E-machine, being one of them in C under Linux using Portable Operating System Interface (POSIX) threads and semaphores.

3.1.2 WCET-aware C Compiler

The WCET-aware C Compiler (WCC) was the first compiler to provide means to reduce the Worst Case Execution Time (WCET) at both, source code and assembly code level. First, Fidge [41] presented techniques to integrate WCET data into the WCC compiler infrastructure. Using these techniques the compiler's low-level IR code is automatically feed into the WCET analyser (aiT static timing analysis tool developed by AbsInt [4]) and the result produces by the WCET analysis tool is re-imported into the compiler infrastructure. Then, Falk and Lokuciejewski [39] completed the implementation of the WCC, a compiler that has a clear notion of a program's worst-case behaviour (combining measurement-based WCET analysis and static program analyses) and applies specialized compiler optimizations to reduce program's WCET. To this end, the WCC's optimizations focus exclusively on those parts of the program that lie on the Worst Case Execution Path (WCEP), the path within program's Control Flow Graph (CFG) that has the maximal WCET. The WCEP may change on the course of an optimization, therefore, it must be recomputed whenever necessary. The WCC's target architecture is the Infineon TriCore TC1796 and TC1797 processors heavily used in the automotive industry.

3.1.3 BIP & FreeRTOS

Behaviour Interaction Priority (BIP) [1] is a component-based framework for building systems by superposition of the Behaviour (B), Interaction (I), and Priority (P) layers. The Behaviour layer consists of a set of components represented by timed automata [6]. The Interaction layer describes possible interactions between atomic components (defined by connectors), providing means for synchronization. The third layer defines priorities between conflicting interactions.

Real-Time BIP (RT-BIP) [2] is the extension of the BIP language for modelling real-time systems together with a real-time engine used for execution. RT-BIP framework, combines an abstract model representing the behaviour of a given application software integrating user-defined platform-independent timing constraints, and a physical model that describes the behaviour of the abstract model by assigning to its actions an upper bound of the actual execution times for a specific platform. Then, based on a time-robust physical model, the real-time execution engine coordinates the execution of the application software to meet the corresponding timing constraints. When robustness cannot be guaranteed, the execution engine can detect time-safety violations and stop execution.

Le Nabec et al. [65] describe the process of modelling real-time legacy code using the BIP framework. To this end, they specify a configurable component pattern, called Real-Time BIP Agent (RT-BIPAgent), that follows a classical template for a real-time task, composed of a start time, a period, a set of input data, a set of output data and a computational function. Then, based on the FreeRTOS platform, executable code is generated from the BIP model. Each RT-BIPAgent models a real-time task and for each connector in the BIP model a FreeRTOS queue is created.

3.1.4 Timed C

Natarajan and Broman [73] proposed an extension of the C programming language, called Timed C, consisting of a set of primitives for defining soft and firm real-time constraints that ease real-time systems' implementation. Authors defined primitives for introducing soft and firm timing delays that can be implemented to define a periodic behaviour, as well as primitives for defining concurrent tasks with communication channels that can be scheduled by an underlying real-time operating system according to the scheduling policy and priority determined through the scheduling primitives. Then, using their source-to-source compiler KTC, a Timed C file is compiled into a target specific C file. The resulting file can then be linked

against POSIX or FreeRTOS to implement the user defined timing and scheduling behaviour.

3.1.5 Real-Time Concurrent C

Gehani and Ramamritham [42] implemented one of the few C languages that incorporates a set of temporal constructs into Concurrent C, a parallel superset of C. Real-Time Concurrent C provides means to specify strict timing constraints through the temporal constructs, which allow delaying program's execution, defining periodicity or specifying deadlines. Whenever the specified timing constrains are not met or compliance of timing constraints cannot be guaranteed, alternative actions can be performed. Given that Real-Time Concurrent C was designed for a UNIX-based implementation of Concurrent C and its compiler is no longer available, Real-Time Concurrent C is considered to be outdated [73].

3.1.6 Time Measurement and Control Blocks

The timing measurement and control blocks [22] are a C++ extension, implemented as a C++ library, that provide to software developers support to add block-level timing annotations into embedded C++ software. Through these annotations it is possible to measure and profile software block's execution timing. In addition, the blocks can also be implemented to control and enforce a specific timing behaviour at run-time. Using the timing control blocks two execution modes can be defined; either the specified block duration is kept precisely by inserting a delay after blocks execution, or execution continues with the next block right after execution of the current block finishes and remaining time until the expected end of the current block (according to the star of block's execution and the specified time limit) is passed to the next block. In order to define a more complex timing behaviour, blocks can be nested. Moreover, blocks are never allowed to run longer than expected (considering the budget from previous blocks) and a timing violation is reported when the block is about to exceed the specified time limit. Time measurement and control blocks have been implemented for bare-metal C++ applications running on an Zynq-7000, using the Global Timer Counter and a Central Processing Unit (CPU) Watchdog, therefore, the solution has limitations regarding its portability.

3.1.7 Timing-aware Recompilation – Analysis

Considering the purpose of the work described in this thesis, this section analyses the presented timing-aware recompilation solutions with a focus on the following aspects: portability, legacy code support, timing enforcement and WCET reduction. Table 3.1 summarizes the analysis on timing-aware recompilation solutions:

Retargetability is an important enabler for a wide adoption of any software tool [36] and is a great concern when it comes to implement a RT legacy software migration solution. However, the analysed timing-aware recompilation solutions have limitations regarding portability. In the solutions based on LET and BIP, both, the E-machine and BIP execution engine need to be retargeted to the corresponding target platform; the same happens with the WCC, which currently only supports the Infineon TriCore target processor; and the Timed C approach also requires its source-to-source compiler to be adapted to the target architecture. Whereas porting Real-Time Concurrent C means implementing a compiler from scratch, since its compiler is no longer available. While the solution presented by Bruns et al. [22] requires the adoption of timer accesses to the new target platform.

Given that the main objective of the presented work is finding a migration path for real-time legacy software, **legacy software** support is another important factor. Although, in theory, any of the analysed solutions is applicable to legacy C code, [80] and [65] are the only ones who consider RT legacy C code and describe the transformation process to LET-based software (compiled into E code and interpreted by the E-machine) and a BIP-based model (executed on top of FreeRTOS), respectively.

On the direction to provide a migration path to real-time legacy code, one of the main contributions of this work consists of providing means for legacy **timing enforcement**. Every analysed solution, except for Falk and Lokuciejewski [39], provides means to enforce a given timing behaviour during execution. Resmerita et al. [80] transform a legacy control application composed of time triggered as well as event triggered tasks with different priorities using the TDL. Moreover, they implemented a tool that provides support on the transformation process. Le Nabec et al. [65] implemented a BIP component pattern that follows the classical template for a real-time task. Through this pattern and the BIP framework it is also possible to describe time and event triggered tasks with different priorities. However, the latter is based on timed automata, which allows more general timing constraints than LET (i.e., lower and upper bounds or time non-determinism). The C programming language extension presented by Natarajan and Broman [73], provides means to describe periodic as well as aperiodic real-time tasks with different

Table 3.1.: Timing-aware recompilation solution analysis. Each of the solutions described in Section 3.1 is analysed according to four aspects: portability, legacy code support, timing enforcement and WCET reduction.

Name	Portability	Legacy Software	Timing Enforcement	WCET Reduction
LET & E-machine [80]	Retargeting E-machine	✓	Time & Event triggered tasks with priorities	-
WCC [39]	Retargeting WCC	-	No support	✓
BIP & FreeRTOS [65]	Retargeting execution engine	✓	Time & Event triggered tasks with priorities	-
Timed C [73]	Retargeting KTC	-	Periodic & aperiodic RT tasks with priorities & soft & firm deadline	-
Real-Time Concurrent C [42]	Retargeting compiler	-	Delays, periodicity & deadlines	-
Time Measurement and Control Blocks [22]	Adapt timer accesses	-	Time budget & deadlines	-

deadline criticality and configurable priority through Timed C primitives. Real-Time Concurrent C provides means to delay program's execution, define a periodic behaviour or specify deadlined. Last but not least, the Time Measurement and Control Blocks provide means to specifying time budget as well as time deadline within source code blocks.

Although **WCET reduction** is not a goal for the described research work, we consider WCET-aware optimization of the generated code is a relevant research line on the hard real-time systems field. Falk and Lokuciejewski [39] presented the first compiler that applies compilation time optimizations to reduce the WCET on the generated code.

3.2 Binary Translation

The first binary translation techniques were developed in the late 1980s for academic researches and commercial products [30]. In 1987, HP developed one of the earliest commercial binary translation systems to migrate source HP 3000 programs to the new Precision Architecture [18]. Followed by the IBM System/370 simulator running on top of an IBM RT(RISC) PC, MIMIC [69]. Later, binary translation techniques were used by many other affiliations. Indeed, in the '90s translators became quite common among hardware manufacturers aiming a migration path for existing software from their Complex Instruction Set Computer (CISC) architecture to Reduced Instruction Set Computer (RISC) architecture (e.g., Andrews and Sand [7], Silberman and Ebcioglu [87], Sites et al. [88], Cmelik and Keppel [32], and Hookway [52]).

Binary translation techniques have been widely studied and developed in the last decades. So, given the great amount of binary translation systems and the interest of this thesis on embedded real-time legacy software migration, just cross-platform translators heeding portability, embedded systems and/or RT applications will be considered in this section.

3.2.1 Binary Translator for Real-Time Applications

When it comes to translate RT legacy code, not just its functional behaviour, but also the timing behaviour has to be preserved. To the authors knowledge, Cogswell and Segall [33] and Heinz [48] are the only ones who considered RT legacy software in their proposed binary translation approach.

TIBBIT

The TIBBIT project, implemented by Cogswell and Segall [33], was the first binary translation approach developed for embedded RT applications that needed to be migrated to a different processor but still maintaining the externally observable timing behaviour. TIBBIT analyses the timing of each instruction in the source binary code and assuming that execution time for every instruction in the source ISA is constant, generates a target binary with an equivalent timing behaviour. To this end, both, the legacy application and the operating system code, are packed in a black box, converted into an equivalent C program, and then statically translated into the target binary format using the GNU Compiler Collection (GCC) retargetable compiler.

To ease and automate the support of multiple source and target ISAs, a binary-to-binary translator generator is provided, Automated Synthesis of TRAnslators (ASTRA). Based on a source/target machine description file ASTRA produces a translator for that source/target combination. The source machine description file contains the timing information which serves as input to preserve the legacy timing.

To provide a timing invariant migration, during the translation process, timing code is inserted into each BB. The RT supervisor uses these timing annotations to maintain a record of the execution timing on the legacy processor. During execution a virtual clock indicating the execution time in the legacy processor is kept updated. Within every loop and before every I/O event, the virtual clock is compared to the real (wall) clock. If the execution is ahead of schedule, the extra time is used to run other tasks until the execution is back on schedule.

Heinz

In 2008, Heinz [48] proposed an approach to preserve the temporal behaviour of RT legacy software when statically translating binary code. To this end, Heinz presents a solutions based on statically computed temporal barriers that preserve the legacy timing behaviour at a synchronization point accuracy level. The synchronization point accuracy consists on preserving the timing behaviour at specific points in program which are considered time sensitive, such as I/O instructions or accesses to shared memory locations.

The proposed solution statically computes a set of delay constants for each program point and selects the appropriate constant at run-time according to the context of a

program point. This way, the delay computation is shifted from run-time to compile-time avoiding the need to keep track of the execution time on the source machine and the corresponding overhead. The context based constant delay approach can not precisely preserve the temporal behaviour. However, the delay constants are computed such that the deviation from the legacy timing is minimized. Moreover, the temporal displacement can be safely bounded to a maximum deviation.

3.2.2 Machine-adaptable Binary Translators

Implementing a binary translator from scratch requires a great effort, and given that binary translation tools are highly dependent on source and target architectures, there is a great interest on machine-adaptable binary translation solutions.

UQBT & UQDBT

UQBT [29] was the first SBT tool designed with portability in mind. UQBT translates the user-level legacy binary into source machine Register-Transfer Levels (RTLs) (M_S-RTL_S). After the analysis phase M_S-RTL_S is lifted into a machine-independent representation Higher-Level Register Transfer Language (HRTL), then down to a target machine RTLs (M_T-RTL_S) and finally the target machine binary code is generated. To handle indirect calls that could not be discovered at static time, the UQBT uses an interpreter.

Based on the static UQBT framework, Ung and Cifuentes [91] implemented the first retargetable DBT approach, UQDBT. From UQBT the front-end is reused in its dynamic version. However, the granularity of decoding is changed from procedure level to BB level. Just like its predecessor, UQDBT separates the system into machine-dependent and machine-independent parts using a machine independent intermediate representation. The legacy binary is first decoded into a source machine RTL instructions (M_S-RTL_S) which are then converted into machine-independent RTL instructions (I-RTL_S). Finally, these machine-independent RTL instructions are converted into target machine assembly instructions. To improve generated code, UQDBT performs generic hot path optimizations that are applicable on different machines.

This two phase translation, eases the portability of the translators to new source and target architectures. Moreover, both translators can be configured to handle multiple source and target machines through specifications that describe the operating

system's conventions and specific machine instruction set's properties. However, the machine adaptability of the translator comes at the cost of performance.

Since UQBT and UQDBT there have been a wide variety of machine-adaptable binary translation tools.

QEMU

Bellard [16] developed Quick EMUlator (QEMU), a well known machine emulator built upon a fast and portable DBT system. QEMU uses Tiny Code Generator (TCG) to translate target source code into a machine independent IR and then translates IRs into host machine code. In order to reduce system overhead, QEMU applies TB [1] chaining, which directly jumps to the next TB without returning control to the execution engine.

DisIRer

DisIRer [55] is a multi-target SBT tool that leverages the GCC infrastructure. DisIRer translates x86 user-mode instructions into GCC's RTL instructions and then translates RTL instructions into GCC's Abstract Syntax Tree (AST). The fact that it is built upon the GCC optimizer and back-end makes the tool cost effective and easily adaptable to multiple targets (those supported by GCC).

CrossBit

CrossBit [98] is a multi-source and multi-target DBT approach, which, as the rest of portable solutions do, uses a machine-independent IR known as VInst to translate user-level binaries from different source ISAs into binaries hosted by the same OS for different target architectures. CrossBit uses profiling information to determine the hot code where machine-independent optimizations are applied. Moreover, just as QEMU does, it also applies BB chaining to further reduce system overhead.

Rev.ng

Based on SBT technique, Rev.ng [40] is a machine-adaptable binary analysis framework that relies on QEMU and Low Level Virtual Machine (LLVM) [2] to perform the binary translation. Rev.ng takes advantage of the core element of QEMU, TCG, to translate user-level instructions of a supported ISA into a machine independent IR.

[1] A TB is the unit of a basic block in QEMU.

[2] LLVM [64] is a compilation framework that provides a source and target independent optimizer, as well as code generation support for multiple ISAs.

Then, instead of generating machine code for the host architecture in emulation mode, QEMU IR is further translated into a higher level IR, LLVM IR. Employing LLVM as a back-end, the generated LLVM IR is translated into host machine code.

3.2.3 Binary Translators for Embedded Systems

Embedded systems are controllers with a dedicated function and limited resources, which often have real-time constraints and contain a significant amount of low-level code dedicated to control either processor integrated or external devices. For this reason, traditional binary translator might not fit into embedded systems and, therefore, specific solutions might be required.

Baiocchi

Baiocchi *et al*.proposed different approaches to adapt DBT to embedded systems with Scratchpad Memory (SPM), which is a single cycle access and low power memory. To this end, Baiocchi et al. [11] present an approach to manage the translated code cache, which is placed on the SPM, by reducing the amount of additional code injected by the translator. This reduces the cost for re-translating application code, and avoiding eviction of frequently executed code. Baiocchi et al. [12] proposed to bound the size of the translated code cache, located also on the SPM, and to reduce the amount of code injected by the translator to control the execution flow, which accounts for about the 70% of the code in the translation cache. Furthermore, [10] presents a code cache spread between SPM, for most frequently used code, and main memory, where code cache reduction techniques described in [12] are also applied. For their investigation they used Strata [84], an open-source infrastructure for building user-level software dynamic translators. Besides being a portable solution that has been implemented for SPARC, MIPS, and x86 architectures, given that their goal is Flash demand paging [8], its focus is on native BT solutions where the source and target architecture are the same.

Guha

Guha et al. also proposed techniques to adapt DBT tools to embedded systems, by presenting in [45] four different techniques to reduce the amount of code cache occupied by exit stubs, a balanced path selection policy, and a selective flushing approach, which are combined with auxiliary code optimization to improve memory efficiency and performance in [44]. Their solutions have been implemented on Pin [67], an open-source dynamic instrumentation tool for Linux platforms on IA32,

EM64T, Itanium, and ARM architectures. Although Pin has been designed with portability in mind, it is not a cross-platform translation tool.

Chen

Chen et al. [25] developed a SBT solution for embedded systems. However, their tool directly migrates ARM binaries to a MIPS-like architecture, without using an IR, which allows applying architecture specific optimization techniques but at the same time hinders translator's portability. Their translation tool was able to migrate user-level ARM binaries without using a run-time emulator or any DBT support.

LLBT

Shen et al. [86] worked out another SBT tool, LLBT, an LLVM [2] based multi-target SBT tool for embedded systems. LLBT translates ARM user-mode instructions into LLVM IRs and then LLVM IRs are translated into machine code for multiple ISAs. The LLVM compiler infrastructure provided LLBT with means for optimization and retargetability. Moreover, in order to make the system suitable for embedded systems, LLBT reduces the size of the address mapping table.

3.2.4 Binary Translation Tools – Analysis

The proposed solution aims to provide a migration path for RT embedded legacy software. Therefore, this section analyses the presented binary translation solutions (static and dynamic) with a focus on the following four aspects: portability, embedded systems, RT code, and system-level code support. Table 3.2 provides a summary of this analysis.

Development cost is one of the main concerns when developing a binary translator, since the implementation of such a system from scratch requires great effort. Furthermore, binary translation tools are highly dependent on the source and target architectures. For this reason, researchers adopted the general approach of portable compilers, where machine-dependent and machine-independent concerns are separated, to provide a **machine-adaptable** binary translator. Cifuentes and Emmerik [29] implemented the first machine-adaptable binary translator, UQBT. Their approach supports multiple source and target machines by using specifications that describe the ISA and OS. Since UQBT, many other static and dynamic machine-adaptable solutions have been implemented. Unlike UQBT, most of them benefit from a retargetable compiler to provide this same property to their binary translator.

Table 3.2.: Binary Translation tool analysis. Each of the tools described in Section 3.2 is analysed according to four aspects we consider essential for a RT embedded legacy software migration solution: machine adaptability, and RT code, embedded systems and system-level code support.

Name	Static/Dynamic	Machine-adaptable	RT legacy Code	Embedded Systems	User-/System-level
TIBBIT [33]	Static	-	✓	✓	System-level
Heinz [48]	Static	-	✓	✓	System-level
UQBT [29]	Static	Source and Target	-	-	User-level
UQDBT [91]	Dynamic	Source and Target	-	-	User-level
QEMU [16]	Dynamic	Source and Target	-	✓	User- and System-level
DisIRer [55]	Static	Target	-	-	User-level
CrossBit [98]	Dynamic	Source and Target	-	-	User-level
Rev.ng [40]	Static	Source and Target	-	-	User-level
Baiocchi [9] (implemented on Strata [84])	Dynamic	Portable (no cross-platform)	-	✓	User-level
Guha [44] (implemented on Pin [67])	Dynamic	Portable (no cross-platform)	-	✓	User-level
Chen [25]	Static	-	-	✓	User-level
LLBT [86]	Static	Target	-	✓	User-level

However, some binary translation approaches can easily be ported to multiple target ISA, but porting them to a different source ISA involves the entire design process of the corresponding front-end. Therefore, we do not consider them multi-source and -target translators, but just multi-target translators.

Binary translation has been successfully applied to port non-RT legacy code. However, when dealing with time sensitive code migration, not just its functional behaviour, but also the timing behaviour of time critical tasks has to be preserved. From the related work, just Cogswell and Segall [33] and Heinz [48] presented a migration path for **RT legacy software**. The former, proposes a instruction level annotation approach that describes the amount of time required to execute the block on the source processor. This way a virtual clock is provided to the run-time system that compares its value to the target clock and enforces an equivalent timing behaviour. This approach is efficient for simple architectures where the execution time of each instruction is predictable. The latter, implements static temporal barriers to reduce the runtime overhead of the delay computation. Based on a WCET analysis tool a set of delay constants are precomputed for each program point and according to the program context the appropriate value is selected at runtime.

Embedded systems often contain a significant amount of low-level code dedicated to control either processor integrated or external devices (e.g. Analog-to-Digital Converter (ADC); serial, Ethernet or CAN controller; sensor/actuator), also known as **system-level code**. However, most of the approaches in the State of The Art (SotA) propose migration solutions for **user-level code**, where the underlying OS's API abstracts the low-level code from the application. Even if the translator proposed by Chen et al. [25] and LLBT [86] were designed to port embedded system software, they do not support system-level code. These binary translators set their focus on OS-based embedded systems, such as smartphones and tablets. **Baiocchi** and **Guha** developed techniques to reduce the memory footprint of DBT tools to fit them on embedded systems. However, their proposed solutions are applied to user-level translation tools. Whereas, Cogswell and Segall [33] and Heinz [48] presented embedded system capable solutions, but unlike previous solutions, they support system-level binaries. On the contrary, QEMU that now supports embedded system-level binary translation, was fist designed for Linux machine emulation.

3.3 Gap Analysis

Based on the literature review, this section maps existing solutions in the area of timing-aware recompilation and binary translation to the scope of this work in order to identify the research gap.

To this end, related work is identified from R1 to R6 (timing-aware recompilation solutions) and from B1 to B5 (binary translation solutions). Then, identifiers are placed in the scope map according to the research are they cover. The following list shows the related work that has been mapped to the scope:

R1 LET & E-machine [80] (see description in Section 3.1.1)
R2 BIP & FreeRTOS [65] (see description in Section 3.1.3)
R3 Timed C [73] (see description in Section 3.1.4)
R4 Real-Time Concurrent C [42] (see description in Section 3.1.5)
R5 Time Measurement and Control Blocks [22] (see description in Section 3.1.6)
R6 WCC [39] (see description in Section 3.1.2)
B1 TIBBIT [33] (see description in Section 3.2.1)
B2 Heinz [48] (see description in Section 3.2.1)
B3 UQBT [29] & UQDBT [91] (see description in Section 3.2.2)
QEMU [16] (see description in Section 3.2.2)
DisIRer [55] (see description in Section 3.2.2)
CrossBit [98] (see description in Section 3.2.2)
Rev.ng [40] (see description in Section 3.2.2)
LLBT [86] (see description in Section 3.2.3)
B4 Baiocchi [9] (see description in Section 3.2.3)
Guha [44] (see description in Section 3.2.3)
LLBT [85] (see description in Section 3.2.3)
B5 Chen [25] (see description in Section 3.2.3)

Figure 3.1 shows the diagram resulting from mapping related work to the scope, which is composed of four research areas: binary translation, where machine-adaptable solutions form a sub-area of research in binary translation; RT software, where RT legacy software is a sub-area in this group; timing enforcement that forms another research area with retargetable timing enforcement solutions as a subgroup of it; and timing validation which is the fourth research area covered on this research work.

On the one hand, in the are of timing-aware recompilation solutions exists, as ***R1*** and ***R2***, that provide means to enforce a specific timing behaviour within RT software. Moreover, both of these solutions describe how their proposed solutions can

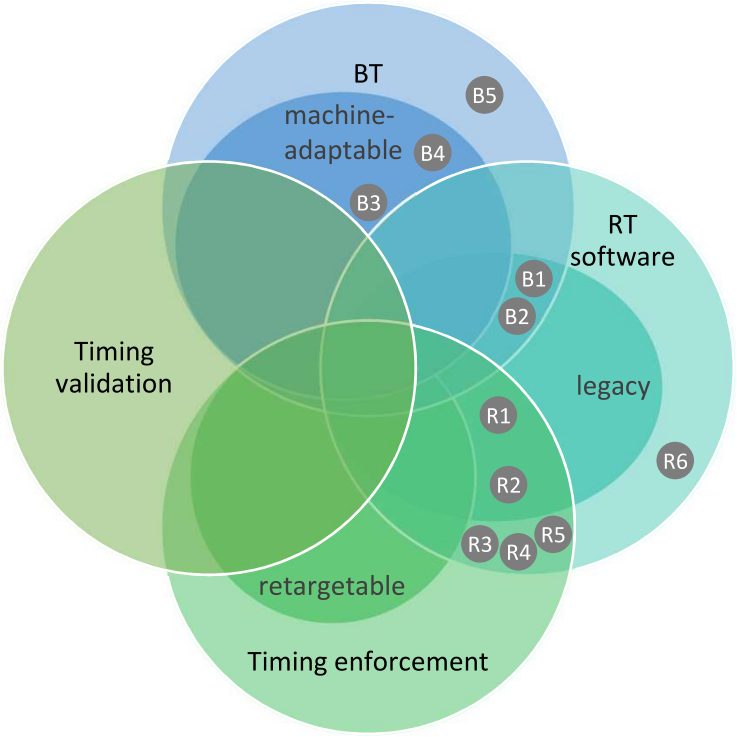


Figure 3.1.: Related work analysis. Mapping related work to the scope.

be implemented on RT legacy software. ***R3***, ***R4***, and ***R5*** are also solution to enforce a specific timing behaviour. However, none of them describe how they can be integrated on RT legacy software. Nevertheless, none of the timing enforcement solutions presents a retargetable solution. Although ***R6*** does not provide means to enforce a specific timing behaviour within RT software, it applies compiler specialized optimizations to reduce program's WCET, which is of great interest for RT software developers.

On the other hand, regarding BT tools, ***B1*** and ***B2*** are the only solutions that provide a migration path to RT legacy software. However, none of these translators is machine-adaptable. In contrast, many machine-adaptable BT tools exist (see ***B3***), but none of them supports a timing-aware binary translation, where not just the functional behaviour, but also the timing behaviour needs to be preserved during the translation process. Concerning embedded system support, solutions exist that aim to reduce the memory footprint of the translator so that it fits in an embedded system (see ***B4*** and ***B5***). Some of them, such as those in ***B4***, are machine-adaptable solutions (although just [85] is a cross-platform tool), whereas ***B5*** is not. However, none of the embedded system solutions supports RT legacy code translation.

4 Thesis Contributions

From the gap analysis (see Section 3.3) we concluded that few solutions exist to port embedded RT legacy software while still preserving its timing behaviour. Moreover, existing solutions have limitations regarding their portability.

In the direction to provide a portable real-time legacy software migration solution, this chapter presents the contributions of this work, and describes how they fit in the scientific scope in order to, at some point, fill the existing gap. The assumptions and constraints concerning the real-time legacy software migration solution are also presented.

4.1 Contributions

The overall goal of this research work is to provide a RT legacy software migration solution based on a existing BT solution, which will be enhanced with a timing enforcement mechanism that at the same time provides means for validating the enforced timing behaviour. All in all the contributions of this thesis are:

C1 A feasibility study of two machine-adaptable binary translators, one dynamic and the other one static, for their use in a RT property conserving legacy code migration process.

C2 A set of portable temporal constructs that provide means to measure the duration of legacy code sections and means to enforce a specific timing behaviour within the legacy software.

C3 A systematic annotation of legacy timing properties into the behavioural legacy code using the temporal constructs, to transform timing properties implicit on the legacy application into explicit timing properties.

C4 A systematic transformation of legacy timing properties into formal timing specifications for their latter use within the timing validation phase.

C5 The integration of the temporal constructs within the binary translation process to achieve a timing-aware binary translation.

By mapping the presented contributions to the scope of this research work, as shown in Figure 4.1, the gap intended to cover through each of the contributions can be appreciated.

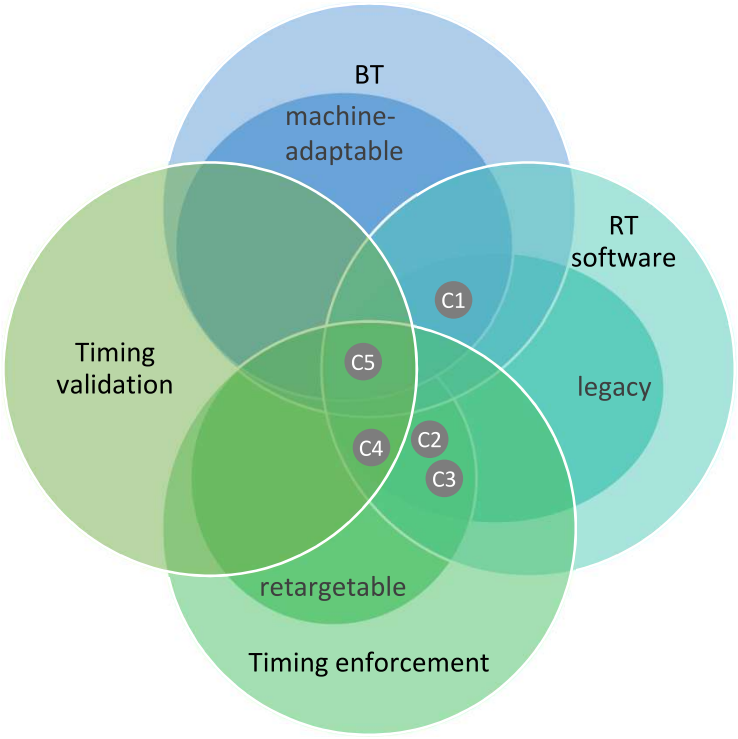


Figure 4.1.: Contributions analysis. Mapping contributions to the scope.

The first contributions (***C1***) works in the are of machine-adaptable BT tools to assess existing solutions with respect to timing. Then, the second contribution (***C2***) provides a retargetable timing enforcement mechanism, which can be systematically applied to RT legacy software through the third contribution (***C3***). ***C4*** contributes in the area of timing validation through the systematically obtained formal timing specifications. Finally, contribution ***C5*** covers every scope area providing a machine-adaptable timing-aware RT legacy software translation with means for timing validation.

4.2 Assumptions & Constraints

The contributions listed in the previous section are tied to the following assumptions and constraints:

A&C1 The new processor (multicore processors are out of the scope in this first approximation) has greater processing capacity (measured in Cycles Per In-

struction (CPI)) than the legacy processor, such that makes affordable the overhead of the timing enforcement mechanism management and code translation overhead. The impact of multicore processors is out of the scope in this first approximation.

A&C2 The legacy source code and toolchain are available, since the proposed timing enforcement mechanism is applied at source code level.

A&C3 The legacy toolchain supports Linux, which is a requirement implicit to the selected binary translation tool.

A&C4 The timing properties of the legacy application are known.

A&C5 The legacy application follows the typical pattern of a reactive control system consisting of a set of periodic tasks (which might have different periods) executed following a static scheduling policy.

A&C6 Legacy platform dependent I/O is not considered, since the presented approach does not provide I/O virtualization between the legacy and new hardware platform.

A&C7 A binary translation framework for the legacy and new ISA is available, since the design and implementation of a new front- or back-end is not considered in this work.

Section 8.2 presents an outlook on future research work to relieve some of these shortcomings.

5 Real-Time Legacy Software Migration

This chapter presents a retargeting solution for RT legacy code. Section 5.1 presents the legacy system model, which is followed by the description of the real-time legacy software migration process in the next sections. Figure 5.1 depicts (from left to right) the real-time legacy software migration process (described in the remaining sections) that ports the RT legacy control software running on top of the legacy hardware platform to a new (different) hardware architecture.

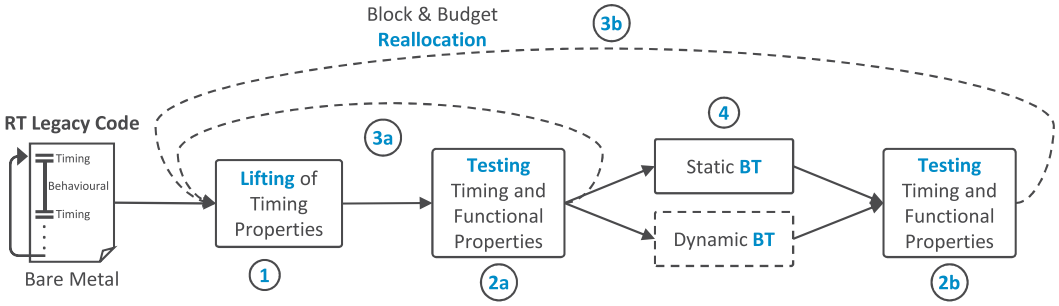


Figure 5.1.: RT Legacy Software Migration flow. Lifting of timing properties (step 1) is described in Section 5.2. Testing timing and functional properties (step 2), as well as block and budget reallocation (step 3) are described in Section 5.3. Timing equivalent legacy software porting (step 4) is described in Section 5.4.

The migration process consists of four main steps. The first step (marked with number 1 in the figure) corresponds to the process of lifting the legacy timing properties (extract legacy timing properties and transform them into formal timing specification), which is presented in Section 5.2. To this end, time measurement and control blocks are annotated within the legacy application. These blocks, which are based on the blocks presented by Bruns et al. [22], provide means to extract the legacy timing properties and enforce them during execution. The annotated code is then tested to check whether it meets the legacy system's timing and functional properties (see number 2a in the figure). To do so, timing properties are transformed into formal timing specifications and compared against time traces, whereas a set of reference values for input state variables and the corresponding output control variables are extracted from the legacy system to check the functional requirements. According to the results obtained, if any of the requirements (temporal and/or

functional) are not met, time control blocks might have to be reallocated and the time budget as well as the timing specifications adjusted, see number 3a in the figure. The timing and functional test procedure as well as the block and budget reallocation process is described within Section 5.3. The next step, number 4, consists of porting the timing annotated binary code to the ISA of the new hardware platform, with a focus on how the static/dynamic translation tools handle the timing annotations (dynamic translation is shown with a dashed line since it was considered as an option, but the static approach was chosen instead). The timing block aware translation process is described in Section 5.4. Finally, after the annotated legacy code has been translated, temporal and functional requirements are tested once again (marked as 2b) and if needed time control blocks reallocated and time budget as well as timing specifications adjusted (marked as 3b). It is worth to mention that the block reallocation as well as the time budget and contract adjustment step might have to be accomplished several times during the migration process.

5.1 Legacy System Model Definition

The RT legacy control system is a computer system that executes a set of periodic tasks according to a predefined static scheduling policy. The following subsections describe through formal notation the main modelling elements in the considered RT legacy system.

5.1.1 Application Model

Table 5.1 shows an example set of tasks with the corresponding timing properties. Such a task set is described through the application model.

Definition 5.1.1. (Application Model). The legacy application A is composed of a set of periodic tasks T, where a task t_i can be represented by a tuple $(p_i, \phi_i, e_i, d_i, cr_i)$, where p_i is the period of the task, ϕ_i specifies a release time (as an offset relative to the start of the period), e_i is an upper bound of the execution time of the task (the WCET of the task can be used as this upper bound), d_i is the relative deadline of the task and cr_i is an identifier of the existence of a section of critical code within the task. A critical code section is set to be a section that generates an exchange of information among application tasks within a distributed real-time control system. Every element in the tuple, except for cr_i, is composed of the value v and the corresponding time unit tu.

5.1.2 Execution Model

The set of periodic tasks T is executed according to a predefined static schedule. Figure 5.2 depicts the execution trace of an example set of tasks.

Definition 5.1.2. (Execution Model). The execution E consists of a hyper-period H, which determines the time after which the task execution patter repeats itself, that is in turn composed of a set of frames $\{f_j\}$. The size of the hyper-period H is determined through the Least Common Multiple (LCM) among all tasks' period, $H = lcm(p_i)$ and the frame size F is determined through the Greatest Common Divisor (GCD) among all tasks' period, $F = gcd(p_i)$. Both, H and F are composed of the value v and the corresponding time unit tu. According to the hyper-period and frame size, the number of frames within a hyper-period is limited to: $max(j) = H/F$.

Each frame f_j is characterized with a set of time slots $\{sl_{j,1}, sl_{j,2}, \ldots, sl_{j,n}\}$, describing each time slot $sl_{j,k}$ as a tuple $(t_{j,k}, s_{j,k}, e_{j,k})$, where $t_{j,k}$ is the task mapped to the slot, $s_{j,k}$ is the start instant of $sl_{j,k}$, and $e_{j,k}$ the end instant of $sl_{j,k}$. Time slots are consecutively ordered so that $\forall k < n : e_{j,k} \leq s_{j,k+1}$.

The function $\alpha : t_i \rightarrow sl_{j,k}$ maps tasks to slots. A task can only be mapped to one slot $sl_{j,k}$ within a frame f_j. However, a task can be mapped to the same slots within different frames. For example, task t_1 can be mapped to $sl_{1,2}$, $sl_{2,2}$ and $sl_{3,2}$, but never to $sl_{1,2}$ and $sl_{1,3}$, since a task can only run once in each frame.

When mapping tasks to slots, it is assumed that if a precedence relation exists among two tasks $t_i, t_l \in T$, such that t_l shares the results produced by t_i, then t_l will never start before t_i has finished execution: $\forall (t_i, t_l) \in T : \alpha(t_i).e_j < \alpha(t_l).s_j$

5.1.3 Example Application

For a better understanding of the presented approach, the whole chapter is drawn on an illustrative example that resembles the typical pattern of reactive control systems. Consider a legacy system consisting of seven tasks $t_i, i = 1, ..., 7$. A task consists of a sequential code block that starts reading input data and its internal state and terminates when it provides the computed results and updates its internal state. Tasks are ordered considering precedence relation and data sharing among them. Through the use of timers and internal counters, tasks run periodically following a static scheduling policy. Tasks t_1, t_2, t_5, t_6 and t_7 have a period of $20\ ms$, whereas t_3 and t_4 have a period of $40\ ms$. Moreover, tasks t_1, t_3, t_4 and t_6 consist of a critical section. Therefore, in order to preserve correctness of the entire system's behaviour,

the instant at which these critical tasks run must be kept equivalent on the migration process (same offset with respect to the start of the period as in the legacy system and minimum jitter among subsequent task instances). The offset of tasks t_1, t_3, t_4 and t_6 is $0, 10, 10$ and $15\ ms$ respectively. On the contrary, for tasks t_2, t_5 and t_7, a variation in their offset and jitter among subsequent task instances does not hinder a correct behaviour of the overall system. However, precedence relation and data sharing among tasks must still be considered. Table 5.1 summarizes this information and completes it with the WCET of each task. Whereas, Figure 5.2 depicts the execution of the example task set.

Table 5.1.: RT legacy application example. Tasks with their corresponding timing information, such as period, offset (if relevant, otherwise Not Relevant (N/R) is shown), an upper bound for the execution time, as well as identification of critical sections are shown.

Task	Period $[ms]$	Offset $[ms]$	WCET $[ms]$	Critical section
t_1	20	0	5	✓
t_2	20	N/R	5	-
t_3	40	10	3	✓
t_4	40	10	3	✓
t_5	20	N/R	2	-
t_6	20	15	2	✓
t_7	20	N/R	3	-

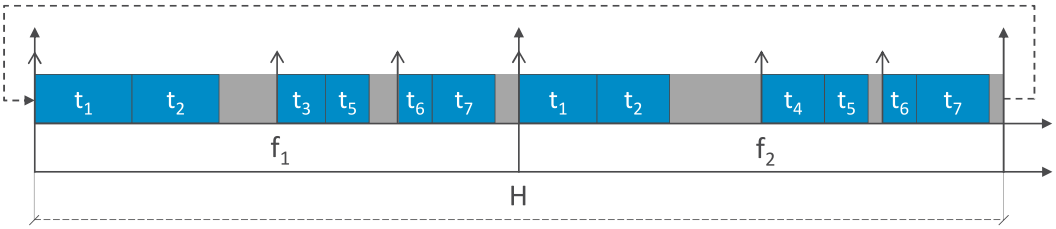


Figure 5.2.: Example execution trace of the RT legacy application example. Execution is composed of two frames that compose the hyper-period.

5.2 Lifting of Timing Properties

The timing property lifting process (see Figure 5.3) consist of three main stages. The profiling phase constitutes the first stage, where expert knowledge is crucial to properly characterize the systems temporal behaviour. As a result of the profiling phase, the behavioural legacy code and the corresponding timing properties are obtained. The next step consist of the annotation of the timing properties into the behavioural code in order to enforce an appropriate temporal behaviour during

execution. To this end, time measurement blocks are substituted with time control blocks (Periodic Execution Time (PET), Forced Execution Time (FET), Budgeted Execution Time (BET) and Period N Execution Time (PNET), described in Section 5.2.2). Finally, the annotated legacy application is systematically transformed into formal timing specifications that will later be used to validate the timing behaviour of the annotated legacy application.

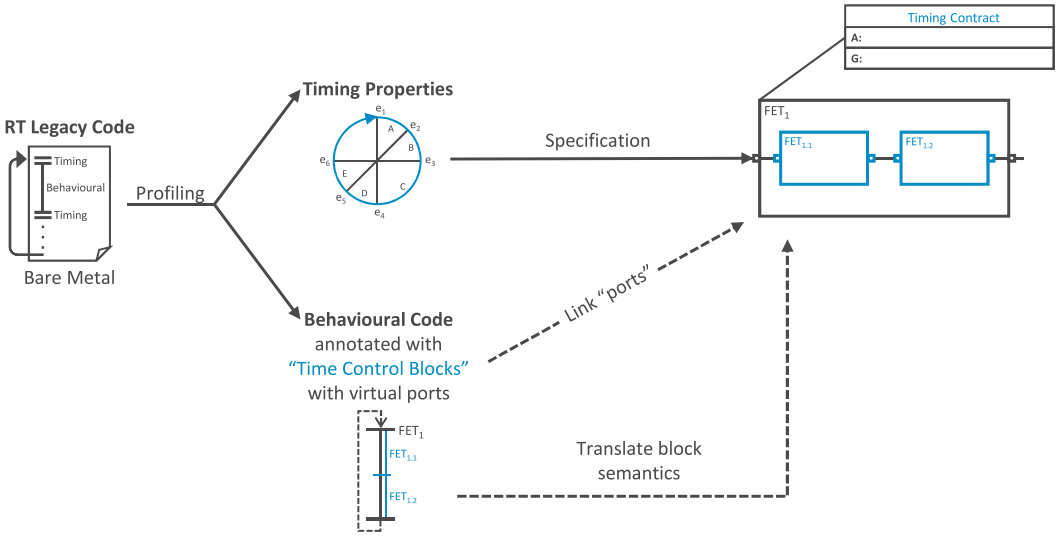


Figure 5.3.: Lifting of timing properties. Profiling phase: extract timing properties and behavioural code from RT legacy code. Annotation phase: annotate behavioural code with time control blocks. Time Specification phase: transform annotated timing properties into timing specifications attached to virtual ports.

5.2.1 Profiling Legacy System

Although the legacy timing properties are considered to be known, this subsection presents an overview of the profiling phase. As depicted in Figure 5.4, the profiling phase, combines code analysis, legacy system's specifications and timing measurements (the Estimated Execution Time (EET) described in Section 5.2.1 can be used to perform timing measurements), which are then evaluated by an expert, to extract the necessary timing information from the legacy system.

Information regarding the period of each task, precedence relation and data sharing among tasks as well as identification of critical sections must be obtained through the analysis of the legacy source code, legacy system's specifications and expert knowledge. Whereas the EET block can be helpful to extract information concerning the WCET and offset of tasks.

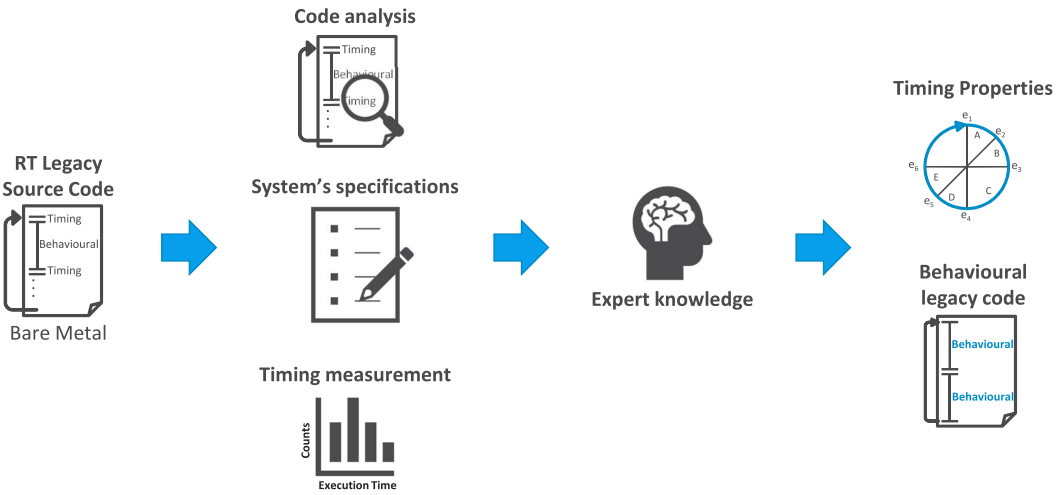


Figure 5.4.: Profiling phase.

In order to systematically annotate the legacy code, first, Algorithm A.1.1 (in Appendix A.1) sorts out the task set T (see Definition 5.1.1) according to the execution model (see Definition 5.1.2), the output consists of the sorted task set T_{SO} (see Definition A.1.1 in Appendix A.1).

Then, Algorithms A.1.2 to A.1.4 (in Appendix A.1) describe the systematic annotation of the execution model with time measurement blocks, EET blocks. The output of the main algorithm (see Algorithm A.1.2) consists of a set of time measurement block nodes MB (see Definition A.1.2).

Example 5.2.1

The example application presented in the previous section (see Section 5.1.3) is annotate with EET time measurement blocks. To this end, the periodic loop must be identified first, which is then wrapped into an EET block. This will be the root node that will provide information regarding the WCET of a frame, which corresponds to the frame size F. Then, each legacy task is wrapped into an independent EET block. Moreover, to compute the offset of tasks identified as a critical section, every task preceding such a critical task is wrapped into another EET block, where the WCET of the wrapped code block corresponds to the offset of the next critical task. Listing 5.1 shows the resulting EET annotated legacy example application.

Time Measurement Block – Estimated Execution Time

The EET block [22] employed in the profiling phase provides means to measure the execution time of the annotated code block and perform a measurement based WCET analysis. For each wrapped code block the execution time duration is computed and stored under a block ID for a latter analysis. After several runs, from the stored data,

```
int main() {
  initialization();
  while(1) {
    XTime_GetTime(&tStart);
    toggle_led(1);
    toggle_led(2);
    do {
      XTime_GetTime(&tNow);
      tEnd = 1.0*(tNow-tStart)/
        (COUNTS_PER_SECOND/1000000);
    } while(tEnd<10000000);//10ms
    switch(i){
    case 0:
      toggle_led(3);
      i=1;
      break;
    case 1:
      toggle_led(4);
      i=0;
      break;
    }
    toggle_led(5);
    do {
      XTime_GetTime(&tNow);
      tEnd = 1.0*(tNow-tStart)/
        (COUNTS_PER_SECOND/1000000);
    } while(tEnd<15000000);//15ms
    toggle_led(6);
    toggle_led(7);
    do {
      XTime_GetTime(&tNow);
      tEnd = 1.0*(tNow-tStart)/
        (COUNTS_PER_SECOND/1000000);
    } while(tEnd<20000000);//20ms
  }
}
```

a: Legacy example application.

```
int main() {
  initialization();
  while(1) {
    EET() {
      EET() {
        XTime_GetTime(&tStart);
        EET() {
          toggle_led(1);
        }
        EET() {
          toggle_led(2);
        }
        do {
          XTime_GetTime(&tNow);
          tEnd = 1.0*(tNow-tStart)/
            (COUNTS_PER_SECOND/1000000);
        } while(tEnd<10000000);//10ms
      }
      EET() {
        switch(i){
        case 0:
          EET() {
            toggle_led(3);
          }
          i=1;
          break;
        case 1:
          EET() {
            toggle_led(4);
          }
          i=0;
          break;
        }
        toggle_led(5);
        do {
          XTime_GetTime(&tNow);
          tEnd = 1.0*(tNow-tStart)/
            (COUNTS_PER_SECOND/1000000);
        } while(tEnd<15000000);//15ms
      }
      EET() {
        toggle_led(6);
      }
      EET() {
        toggle_led(7);
      }
      do {
        XTime_GetTime(&tNow);
        tEnd = 1.0*(tNow-tStart)/
          (COUNTS_PER_SECOND/1000000);
      } while(tEnd<20000000);//20ms
    }
  }
}
```

b: Legacy example application annotated with timing measurement block (EET).

Listing 5.1: Llegacy example application and the resulting time measurement block (EET block) annotated legacy example application.

the average, standard deviation, 99%-quantiles and maximum observed duration are computed, which are also represented in a histogram.

5.2.2 Legacy Timing Enforcement

Once the timing properties (i.e., precedence relation and data sharing, period, offset, and WCET) have been extracted through the profiling phase, the legacy timing behaviour must be enforced. The proposed time control solution is based on a block-level source code annotation approach. According to typical patterns on RT control systems, four different annotation blocks have been defined: PET, FET, BET and PNET.

- The PET block is used to implement the periodic execution of a frame, the argument passed to this block corresponds to the frame size (F).
- The FET block allows preserving a specific offset for tasks and a minimum jitter among subsequent task instances (although there might always be some jitter incurred by control block management, the underlying OS, and the hardware platform itself). Therefore, every task preceding a critical task must be encapsulated into a FET block.
- The BET block is used to allocate a time budget to tasks without enforcing a specific duration. This block might be used to wrap tasks preceding others without any critical code section.
- The PNET block allows allocating a time budget to tasks that run with a period greater than that specified in the PET block. Combining the period and offset arguments of this block, tasks can be mapped into different frames.

To enforce the legacy timing behaviour, the legacy behavioural code (see Listing 5.2.a where the code used to implement the static scheduling policy described in Definition 5.1.2 has been removed) is annotated with time control blocks.

Algorithms A.2.1 to A.2.4 (see Appendix A.2) describe the systematic transformation of the legacy system's model into time control block based annotated legacy code. The input to the main algorithm (Algorithm A.2.1) consists of the frame size F and a set of sorted tasks where every time element has the same time unit $T_{SO_{unit}}$ (see Definition A.2.1). The output resulting from Algorithm A.2.1 consists of a set of time control block nodes CB (see Definition A.2.2).

Example 5.2.2

The example legacy application presented in Section 5.1.3 is annotated as follows. The root node is a PET block with period set to 20 ms, which is the GCD of all tasks' period. Then, each task is wrapped into a BET or PNET block according to its period. Tasks with a period

equal to that defined through the PET block are wrapped into BET blocks, whereas tasks with a greater period are mapped into PNET blocks. The budget for either block (BET or PNET) is determined by the WCET of the task it contains. Whereas the period and offset parameters of PNET blocks are determined according to the period of the task they wrap. For the example application, tasks t_3 and t_4 have a period two times greater than that defined on the PET block, therefore, the period argument is set to 2, whereas the offset parameter is set to 0 in the PNET block that wraps t_3 and to 1 in the block containing t_4. This way t_3 and t_4 will run every two periods starting at period 0 and period 1 respectively. Finally, in order to preserve a specific offset and minimum jitter among subsequent task instances for critical tasks, every BET or PNET block preceding a block wrapping a task marked as critical section must be wrapped into a FET block. Therefore, BET blocks corresponding to t_1 and t_2 are wrapped into a FET block. The duration of this FET block is set to 10 ms so that the next tasks marked as critical preserve their offset. The PNET blocks and the BET block corresponding to t_5 are wrapped into another FET block. The upper limit of PNET blocks that can be wrapped in a FET block is determined by the period argument of the PNET blocks, which has to be equal for every PNET block within a FET, while the offset has to be different for each PNET block within a FET. To preserve the offset of the following critical task (t_6), the duration of the FET block is set to 5 ms. The resulting annotated code is shown in Listing 5.2, together with the behavioural legacy code, and the corresponding tree diagram in Figure 5.5.

```
1  int main() {
2    initialization();
3    toggle_led(1);
4    toggle_led(2);
5    toggle_led(3);
6    toggle_led(4);
7    toggle_led(5);
8    toggle_led(6);
9    toggle_led(7);
10 }
```

a: Behavioural legacy example application.

```
1  int main() {
2    initialization();
3    PET1(20 ms) {
4      FET2(10 ms) {
5        BET6(5 ms) {
6          toggle_led(1);
7        }
8        BET7(5 ms) {
9          toggle_led(2);
10       }
11     }
12     FET3(5 ms) {
13       PNET8(3 ms, period 2, offset 0) {
14         toggle_led(3);
15       }
16       PNET9(3 ms, period 2, offset 1) {
17         toggle_led(4);
18       }
19       BET10(2 ms) {
20         toggle_led(5);
21       }
22     }
23     BET4(2 ms) {
24       toggle_led(6);
25     }
26     BET5(3 ms) {
27       toggle_led(7);
28     }
29   }
30 }
```

b: Time control block annotate legacy example application.

Listing 5.2: Behavioural legacy example application and the resulting time control block annotated legacy example application.

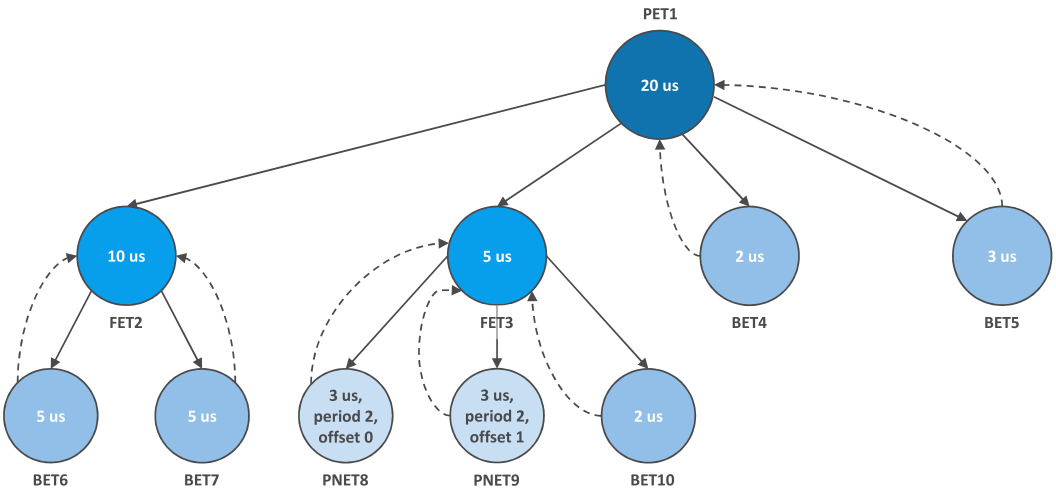


Figure 5.5.: Time control blocks' behaviour and nesting. Description of execution time control blocks' behaviour and nesting through the RT legacy application example (see Section 5.1.3). PET1 is the root node. It has four child nodes: FET2, FET3, BET4 and BET5. PET1 manages the timing budget passed across BET4 and BET5 siblings. FET2 has two child nodes (BET6 and BET7) and manages the time budget passed across them. FET3 has three child nodes, two PNET blocks (PNET8 and PNET9) and a BET block (BET10). The timing budget passes across the sibling PNET and BET blocks is managed by their parent node, FET3.

In the following, time control blocks' runtime behaviour is described. For a better understanding, Figure 5.6 shows the structure and functionality of timing control blocks through an example execution of the application presented in Section 5.1.3.

Time Control Blocks – Periodic Execution Time Block

The PET Block enforces a periodic execution of the wrapped code block. The only argument passed to this block specifies the execution period of the wrapped code block (P_{main}). As shown in Figure 5.6, the PET block inserts a delay at the end of its execution in order to consume the remaining time (if any) and maintain the block's periodicity. On the contrary, if the block takes longer than expected leading to a period violation, a user defined error handling routine takes place. There should only exist one PET block in the whole legacy control application, which will always be the root node.

Time Control Blocks – Forced Execution Time Block

The FET Block enforces a concrete duration (specified in its only input argument) for the wrapped code block. Taking a look at Figure 5.6 it can be seen that, as it

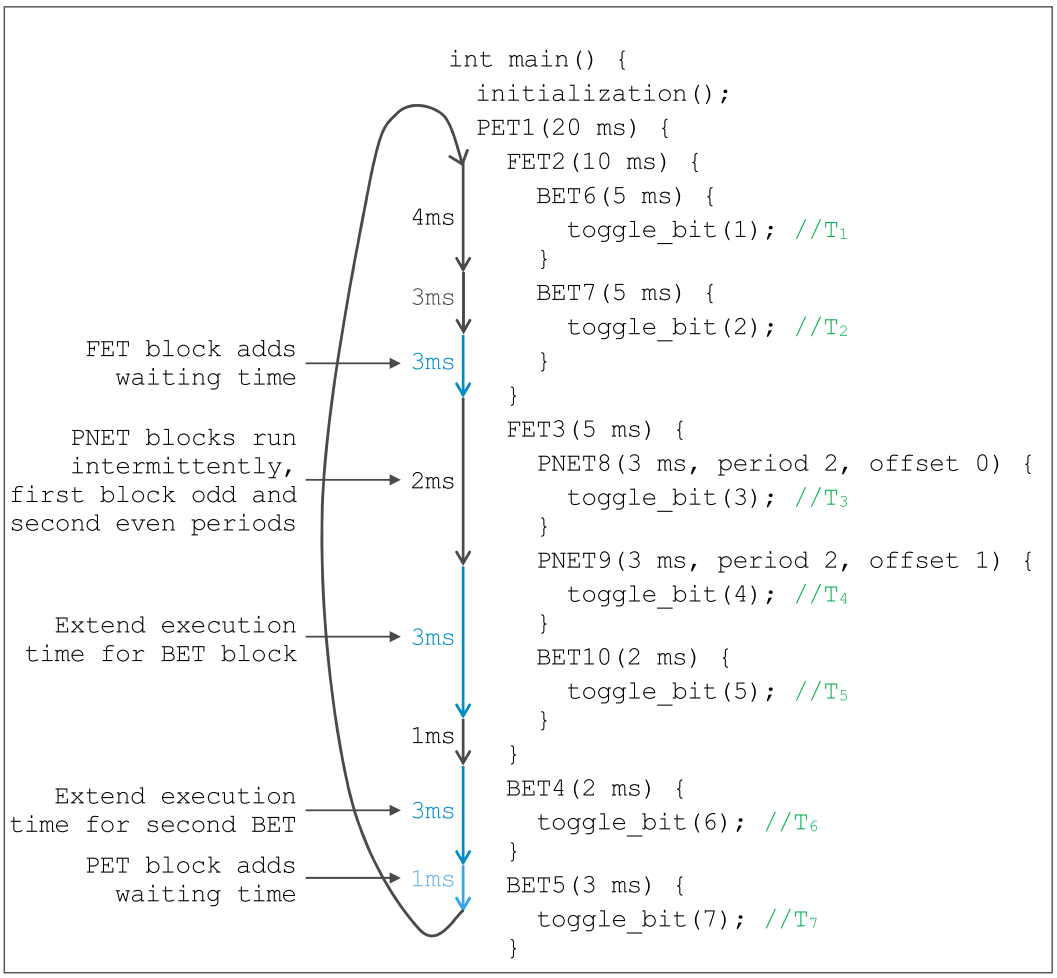


Figure 5.6.: Time control blocks' structure and functionality. Description of execution time control blocks' structure and functionality through the RT legacy application example (see Section 5.1.3). PET block enforces a periodic execution. FET enforces a concrete duration. BET defines an upper bound duration and passes remaining time to next sibling BET or PNET blocks. PNET defines an upper bound duration for a code block that runs every N^{th} period starting at an specific period (according to its offset).

is done in the PET block, extra duration at the end of the block will be consumed (delay insertion). However, if the specified duration is exceeded a user defined error handling routine takes place. FET blocks are always defined within another FET or PET block, therefore, they always have an indirect or direct parent PET block respectively. The use of FET blocks is unlimited; nevertheless, the temporal overhead the block management structure entails should be taken into consideration.

Time Control Blocks – Budget Execution Time Block

The BET Block defines an upper bound duration for the wrapped code block. A BET block should always be defined within another BET or a FET or PET block. Therefore, a BET block will always have a direct (if it is defined within a FET or PET) or indirect (if it is defined within another BET) parent PET or FET block. As shown in Figure 5.6, the time remaining at the end of a BET block execution is passed to the next BET or PNET block (described latter) with the same next (direct or indirect) parent FET or PET block, whatever comes first. This parent block is in charge of managing the execution time budget. However, as FET and PET blocks have a fixed duration, BET blocks can only use the remaining time budget of earlier finished same level BET or PNET blocks. The use of BET blocks is unlimited; nevertheless, the timing overhead introduced by the block management structure should be taken into consideration.

Time Control Blocks – Period N Execution Time Block

The PNET Block accepts three input arguments. The first one determines an upper bound duration for the wrapped code block, the second one determines the N^{th} period at which the blocks is activated, whereas the third one determines the offset of the wrapped code block in periods with respect to the start of the execution. Combining the period and offset arguments, tasks can be mapped to different frames (i.e. N^{th} period 3 and offset 1 means that the block will run every three frames starting with the first run at the second frame due to 1 period offset). A PNET block must always be used within a FET or PET block and remaining time at the end of a PNET block is passed through the following BET blocks (see Figure 5.6). Therefore, a PNET block will always have a direct parent FET or PET block, which will managed the execution time budget among sibling PNET and BET blocks. As in previously described control blocks, a user defined error handling routine takes place if the time budget is exceeded. The use of PNET blocks is unlimited, however, PNET blocks within the same parent FET or PET block should be defined combining the second and third parameters in such a way that they will never run at the same time. Moreover, the temporal overhead of the block management structure should also be considered.

5.2.3 Extract Timing Specifications

The annotated legacy application is systematically transformed into formal timing specifications. To this end, the timing specification language described in Section 2.3 is used, which follows a contract-based approach. This timing specification language was defined within the MULTIC project [14].

The MULTIC project assumes systems to be built from components (see Figure 5.7), which depending on the design context, may represent software functions, hardware elements or any other part of a system. Components interact with the environment (including other components) through a set of ports, which are linked through connectors. A connector can either be simple, when it takes no time to transport a value between the connected ports, or complex, when it represents a physical transmission medium that comprises a complex behaviour. In fact, a complex connector is a component itself. Within this context, timing specifications are defined over component interfaces, the ports of components, since any behaviour in the component model is only observable at the component ports.

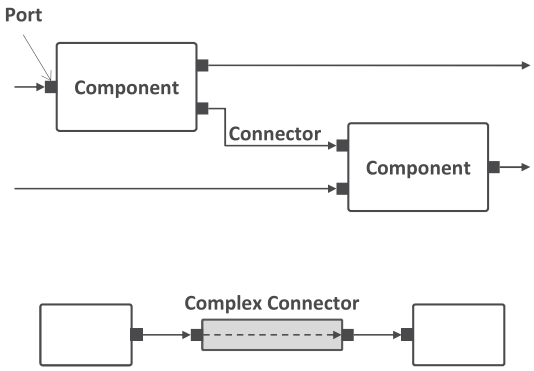


Figure 5.7.: General Component Model.

Based on the general component model, the MTSL described in Section 2.3, specifies ports as follows:

Port :: *PortName* | *ComponentName* '.' *PortName*

Timing specifications about components are expressed in terms of contracts. A contract states on the one hand, assumptions about the components environment and the behaviour that the component's implementation must guarantee, considering the component is used in a context where the assumption about the environment is accomplished, on the other. Contracts are expressed by instances of the MULTIC timing specification language [14] described in Section 2.3. Figure 5.8 shows an

example of a contract, where the assumption(s) (denoted with an A) specify the context in which the component should be executed to accomplish with the behaviour stated as a set of guarantee(s) (denoted with a G). Therefore, the example contract states that assuming that an event occurs on Entry port every 20 ms, whenever an event is observed on Entry port, an event will be observed on Exit port within 0 to 20 ms.

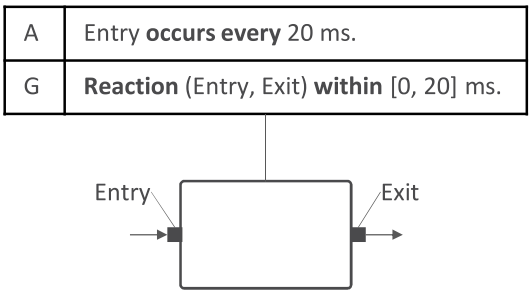


Figure 5.8.: Contract example.

In order to prove correctness of the defined contracts, a Virtual Integration Test (VIT) should be accomplished, which checks two conditions: compatibility and refinement. The former, consist of verifying whether two (or more) components can be put together without violating any of the contracts. That means that two components are compatible if the assumption about the environment of a component are not violated by another component connected to this component. The latter verifies whether the composition of a set of sub-components satisfies the contract(s) of the parent component. This means that the guaranteed behaviour of a set of sub-components must refine the behaviour guaranteed by the parent contract, within an environment that complies with the parent contract assumption. Both the compatibility and the refinement check reason about components of a particular model as well as about components from different viewpoints or abstraction levels (different models). Model artefacts within a viewpoint at a certain abstraction level are allocated to artefacts of other viewpoints, and artefacts at a lower abstraction level are considered to realize those of a higher abstraction level. In this sense, a mapping, which defines an interface between artefacts of models from different viewpoints or abstraction levels, provides the foundation to reason about refinement of the specification of an observing model by the specification of the observed model. Thus, the VIT checks for consistency of models of the system from different viewpoints and abstraction levels meaning that the contacts of the observed model are a valid refinement of the contracts of the observing model.

Although VIT could ideally be achieved by formal methods, this task can become close to impossible. Therefore, the MULTIC-tool provides an alternative through a

simulation based approach. Simulation based VIT is not capable of guaranteeing completeness of the test, but can be used instead for functional testing[1]. To this end, the MULTIC approach assumes that a SystemC simulation model can be derived form a SysML component model, which consists of generators and observers according to the specified contracts. Based on this simulation model a simulation-driven VIT can then be executed.

Based on the MULTIC approach, the RT legacy software migration solution describes each time control blocks as a component. Within each component, virtual ports are defined, at which events are observable. Moreover, component-to-component connections are done according to a causality order. Consistency of contracts is given by constructions, whereas to check compatibility and composition of contracts a VIT should be accomplished.

Algorithms A.3.1 to A.3.5 (see Appendix A.3) describe the systematic transformation of the time control block annotated code into a component-contract structure. The input to the main algorithm (see Algorithm A.3.1) consist of a set of control block nodes $CB = \{cb_m\}$ (see Definition A.2.2), whereas the output consists of a set of component nodes $CO = \{co_m\}$ (see definition A.3.1) with their corresponding contract in the form of assumptions A and guarantees G.

The annotated legacy example application presented in Listing 5.2, which follows a tree diagram as depicted in Figure 5.5, is transformed into a set of component nodes with their corresponding contract. The resulting component-contract structure is shown in Figure 5.9.

5.3 Testing, Reallocation & Adjustment

As a result of the lifting process, the formal timing specifications and the annotated legacy application are obtained. To check that timing and functional requirements are still met after the lifting process, the annotated legacy application is compared against the original legacy application. Therefore, the control block annotated legacy application is executed on the legacy hardware platform to check whether the functional and timing traces meet the reference values. To test the timing properties, the annotated application, the formal timing specifications and the MULTIC tooling [15] are needed. Whereas to test the functional properties, a set of reference values for input state variables and the corresponding output control

[1] Formal methods such as [43, 31] guarantee completeness of the test and should be used in a combination with simulation based testing.

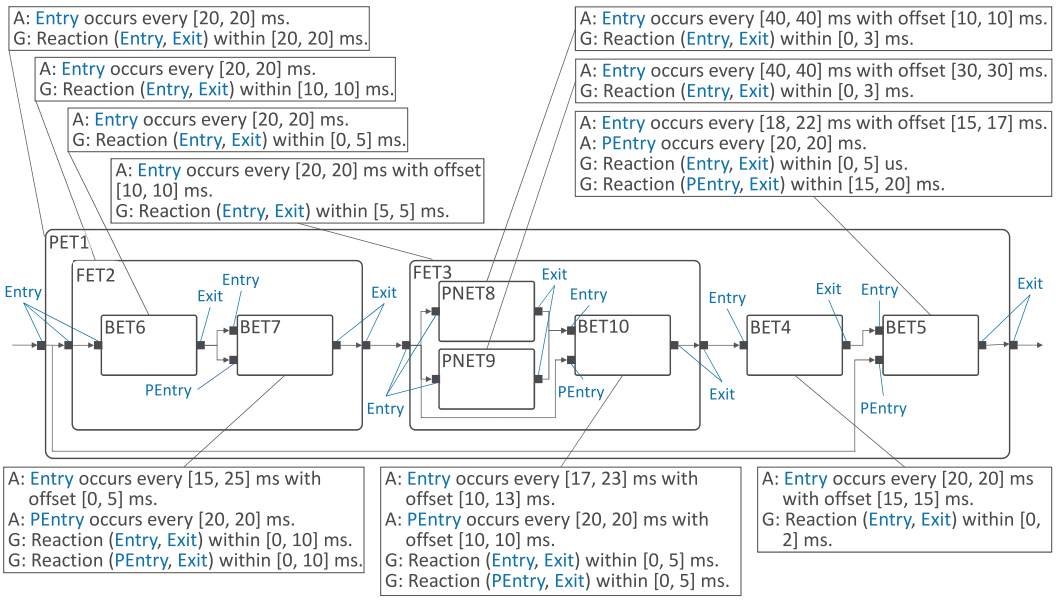


Figure 5.9.: Component-contract structure. Description of component-contract structure through the RT legacy application example (see Section 5.1.3). Each component has its corresponding contract, composed of assumption(s) and guarantee(s), which are based on MTSL repetition and causal reaction patterns. Port names are shown in blue colour.

variables, as well as the annotated legacy application are needed. If the test results with unsatisfied timing or functional requirements, it might be necessary to reallocate and/or adjust either time control blocks, formal timing specifications or both of them.

5.3.1 Testing Timing Properties – MULTIC tool

Figure 5.10 depicts the process of testing the timing properties. The annotated legacy application runs on the legacy hardware platform. As described in the previous sections (see Section 5.2.2) time control blocks generate time traces on runtime. These time traces are then compared against formal timing specifications using MULTIC tool, which allows expressing timing requirements and provides means for their validation and verification through a simulation based method based on the SystemC [51] simulation framework. Timing specification are expressed in terms of contracts through the pattern-based MTSL (see Section 2.3). According to the specified contracts, the tool generates for each specification pattern two automata; a generator, which produces every trace adhered to the specification, and a monitor that recognizes every trace adhered to the specification. Generators and monitors are implemented in C++ code for their inclusion in the simulation framework.

As a result, the tool provides information regarding compliance of system's timing behaviour to timing specifications.

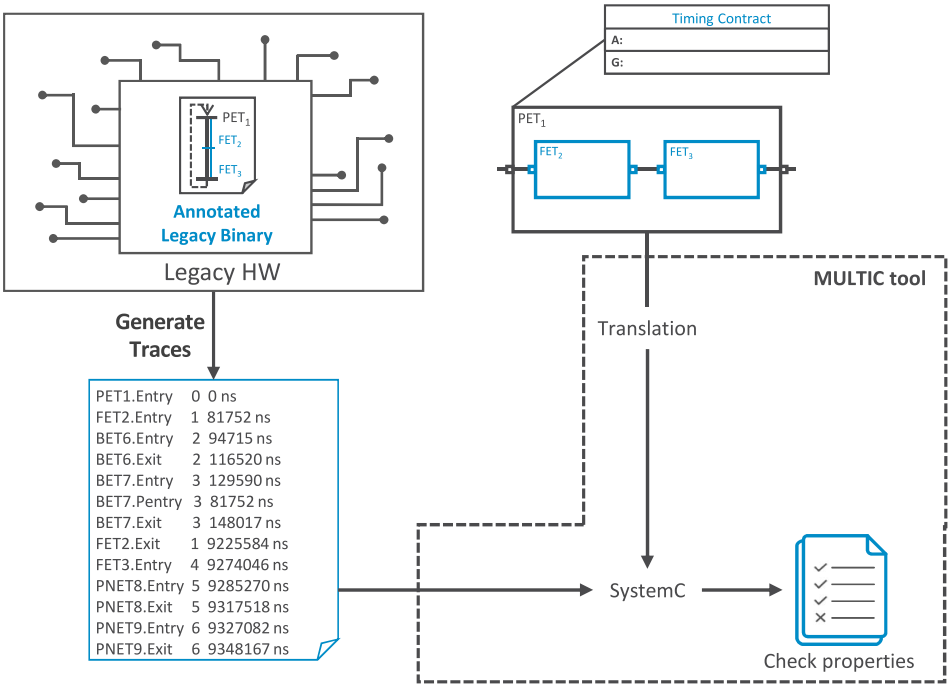


Figure 5.10.: Description of the process for testing timing properties.

5.3.2 Testing Functional Properties

Figure 5.11 depicts the process of testing the functional properties. For the functional test, the reference input/output traces need to be obtained first. To this end, the legacy application is executed with a set of reference input values for state variables and the corresponding reference output values are obtained for the control variables. Once the input/output reference values have been collected, as it is done for testing the timing properties, the annotated legacy application is executed on the legacy hardware platform. During execution, instead of observing state variables, the annotated legacy application is feed with the reference input values (for state variables) and the output values obtained for the control variable are compared against the expected output results, the reference output traces.

5.3.3 Time Control Block Reallocation/Adjustment

According to the timing and functionality test result, a time control block reallocation and budget adjustment process might be necessary. During the lifting process, source

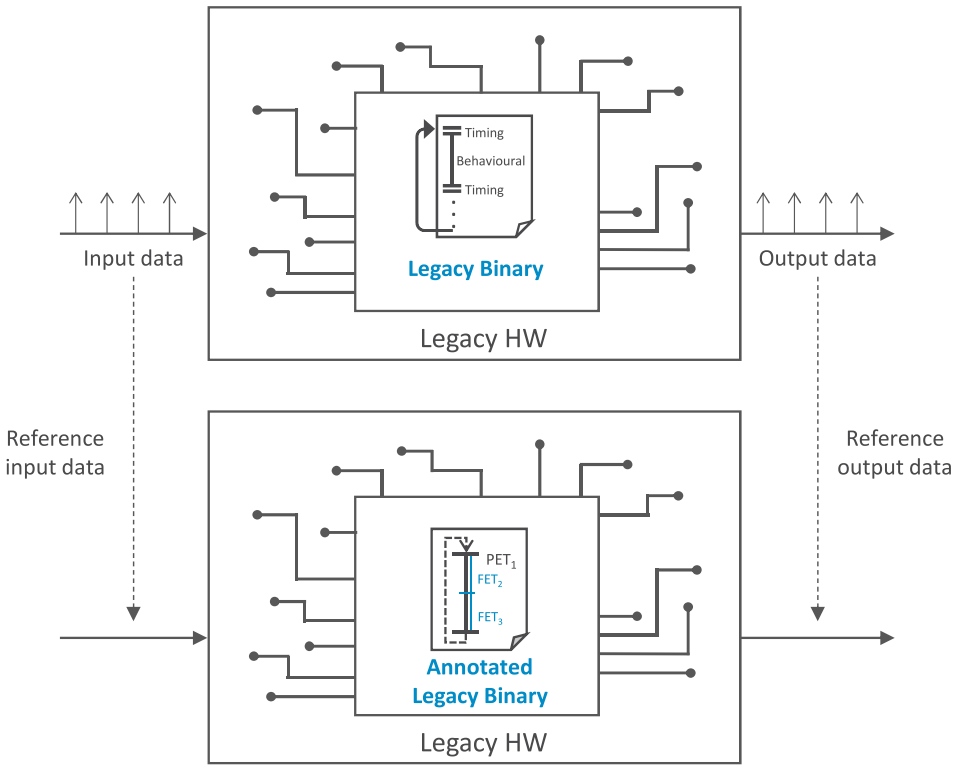


Figure 5.11.: Description of the process for testing functional properties.

code is extracted, modified and rearrange. In this reverse engineering process, the functional and timing behaviour of the legacy application might have been changed. Therefore, if the timing and/or functional test results is not successful (i.e., the allocated budget is not enough, the offset before critical section tasks is not appropriate, the task precedence relation has been corrupted), time control blocks need to be reallocated and budgets adjusted accordingly, while still ensuring an appropriate timing behaviour.

5.3.4 Formal Timing Specification Adjustment

Systematically generated timing specifications (see Section 5.2.3 for information on how timing specifications are generated) describe an ideal timing behaviour where jitter is not considered. However, in a real scenario the timing behaviour is not ideal and timing deviations exist as a consequence to the overhead of time control block management or binary translation, as well as deviations caused by the underlying OS or hardware platform itself. As a consequence, formal timing specifications might need to be adjusted accordingly, while still ensuring an appropriate timing behaviour.

Moreover, every time formal timing specifications are adjusted, it is necessary to prove correctness of contracts through a simulation-based VIT.

During the lifting process it might be necessary to repeat the temporal a functional property test, the control block reallocation and the budget and contract adjustment process several times, until all legacy system's requirements are met, temporal as well as functional requirements.

5.4 Timing Block Support within Binary Translation

The RT legacy software migration approach differs depending on the chosen porting system. The static binary translation tool is a user-level translator that implements the translation at the ABI-level, whereas the dynamic binary translation tool is a system-level translator. Although the dynamic translation tool is a system-level translator, many of the legacy hardware peripherals, such as timers, are emulated. Therefore, it should be considered that the translated legacy application would accesses emulated peripherals instead of accessing target peripherals.

5.4.1 Static Binary Translation based Timing Block handling

The static binary translation tool is a user-level translator that implements the translation at an application-level. As described in the lifting process, the legacy source code is annotated using timing control blocks. The annotated legacy code is then compiled for the legacy hardware platform. To exploit the ABI in the translation process, the static binary translation tool requires a statically linked Linux binary as input file, therefore, a Linux toolchain is used for the compilation process. Then, the translator leverages the ABI to reroute accesses to the underlying hardware by linking the legacy Linux binary translated for the target ISA against the target hardware platform libraries. Figure 5.12 depicts the described static binary translator based timing control block handling. Whereas, Figure 5.13 depicts the run-time architecture of the annotated and statically translated application running on the new hardware platform.

5.4.2 Dynamic Binary Translation based Timing Block handling

The dynamic binary translator based migration solution implements the translation at a system-level. As described in the lifting process, the legacy source code is annotated

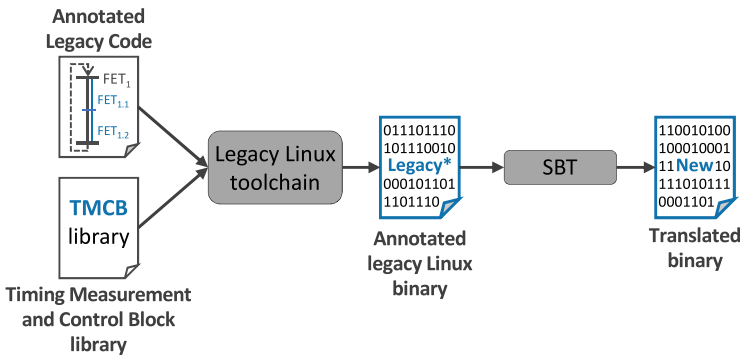


Figure 5.12.: Description of the static binary translator based block handling.

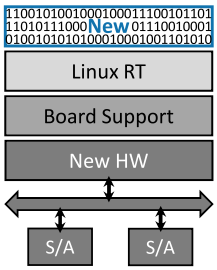


Figure 5.13.: Runtime architecture for statically translated binary running on the new hardware platform.

using time control blocks. However, the dynamic binary translator reroutes timer accesses to the emulated timer. Therefore, the legacy code is annotated with time control blocks, but blocks' management and functionality is implemented at the translation time. The block annotations have no functionality and serve to detect annotation points at translation time. The empty control block annotated legacy code is compiled for the legacy hardware platform and runs on top of the dynamic translator on the new hardware platform. During translation, annotation points are detected and blocks' functionality and management (as described in Section 5.2.2) is implemented. To this end, the DBT tool has to be adapted and linked against the timing measurement and control block library during compilation for the new hardware platform. Figure 5.14 depicts the described dynamic binary translator based block handling. Whereas, Figure 5.15 depicts the run-time architecture of the annotated legacy application running on the new hardware platform on top of the DBT tool.

As it is done after the lifting process, after translating the annotated code temporal and functional properties need to be tested on the new hardware platform. If the test results are unsuccessful, time control blocks might have to be reallocated and/or the assigned time budget and/or timing specifications adjusted (see Section 5.3 for more information).

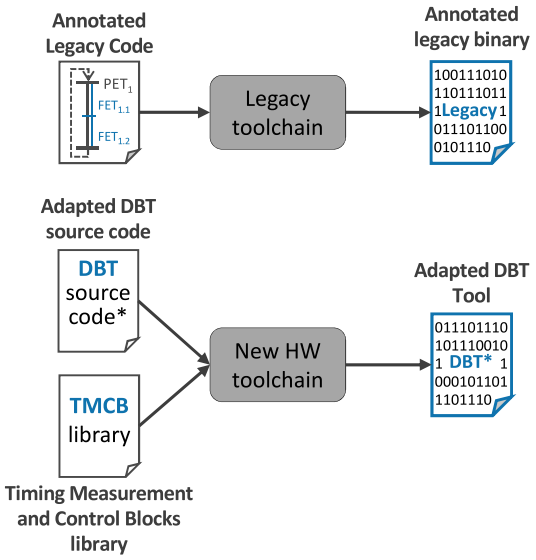


Figure 5.14.: Description of the dynamic binary translator based block handling. Legacy code is annotated with structural blocks. The DBT tool is adapted to implement timing control blocks' functionality and management (at annotation points) during translation.

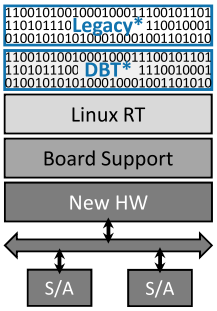


Figure 5.15.: Runtime architecture for the empty control block annotated legacy binary running on the new hardware platform on top of the adapted DBT tool.

6 Implementation

The implementation chapter explains how the aforementioned theoretical model could be implemented using current software and hardware technology. To this end, the RT legacy software migration approach described in Chapter 5 has been implemented as a migration path for RT legacy software running on ARMv7-A to an Intel Atom processor. Based on the binary translation tool analysis on the related work chapter(see Section 3.2.4), two candidate tools, QEMU machine emulator and Rev.ng reverse engineering framework, have been selected to translate code from one ISA to another. These translation tools where mainly selected for being source and target machine-adaptable open source solutions. During translation, the legacy timing behaviour is preserved through a block level source code annotation approach based on a refinement, as described in Section 5.2, of the time measurement and control blocks presented by Bruns et al. [22].

Therefore, this chapter describes first the development platforms (see Section 6.1) and the translation tools with their corresponding translation process (see Section 6.2). Then, in Section 6.4 the implementation of time measurement and control blocks is described. Section 6.4.1 describes how the timing measurement block is integrated into the static and dynamic binary translation process for its use in the feasibility study. Finally, Section 6.6 describes how timing control blocks are integrated on the static binary translation tool to implement the timing contract aware static binary translation.

6.1 Development Platforms

As a proof of concept, the RT legacy software migration approach described in previous chapter (see Chapter 5) has been implemented to port ARM Cortex-A9 legacy software to an Intel Atom processor. The main reason for the selection of these processors has been the support of the ARM-x86 source-target architectures on both binary translation tools, QEMU and Rev.ng. Therefore, the Xilinx Zynq-7000 System on a Chip (SoC) ZC702 evaluation kit and the MinnowBoard Turbot Dual-Core board have been selected as source and target platforms, respectively.

6.1.1 Xilinx Zynq-7000 SoC ZC702

The Zynq-7000 SoC provides software and hardware programmability integrating and ARM multicore processor (Dual-core ARM Cortex-A9 MPCore) and a Field Programmable Gate Array (FPGA) on a single chip. The programmable logic provides flexibility to the SoC, allowing its adoption to a broad variety of embedded applications. Taking advantage of the flexibility of the SoC, both the ARM Cortex-A9 processor as well as the FPGA, implementing a MicroBlaze processor, have been used in this work. The ARM Cortex-A9 MPCore is a 32-bit performance and power optimized multi-core processor implementing the ARMv7-A architecture. Whereas MicroBlaze is a 32-bit soft microprocessor core designed to be implemented on Xilinx FPGAs.

6.1.2 MinnowBoard Turbot Dual-Core

The MinnowBoard Turbot is an open source hardware platform integrating a dual-core Intel Atom E3866 SoC. The MinnowBoard Turbot is a reference design in many embedded products on market today. The Intel Atom E3866 is a 64-bit ultra-low-voltage processor implementing an x86 architecture.

6.2 Translation Tools

From the binary translation tool analysis, a dynamic and a static binary translation tool were selected. The former, QEMU, is an open source machine emulator portable to multiple source as well as target ISAs. Although it was first designed for Linux machine emulation, it currently supports embedded system-level emulation too. Rev.ng is a fully open source reengineering framework that provides means to lift Linux binaries to LLVM IR and recompile them for a different architecture. Moreover, it provides means to perform instrumentation and run various (provided or custom) analyses.

6.2.1 QEMU

The core element in QEMU [16] is its code generator, TCG, which is responsible for the dynamic translation of target source code into host machine code. As a machine-adaptable DBT, the TCG adopts the general approach in portable compilers.

Therefore, the source code TBs (the unit of a basic block in QEMU) are first translated into tiny code instructions, a machine independent IR, and then this IR code is further translated into target machine code. Once translated, the TBs are stored in the code cache to be reused in future runs. TB caching reduces translation overhead since the time spent on code translation is reduced. For the sake of simplicity, when the code cache overflows, all stored TBs are removed. Moreover, to avoid returning control from the code cache to the emulation manager and back again to the code cache, QEMU chains consequentially executed TBs. As an example, after the execution of TB1, as there was no chaining, execution returns to the emulation manager. In that case, the next TB, TB2, has to be found, generated (if target machine code for this TB is not available), executed and chained to TB1. This way, the next time TB1 is executed TB2 will follow the execution without returning control to the emulation manager.

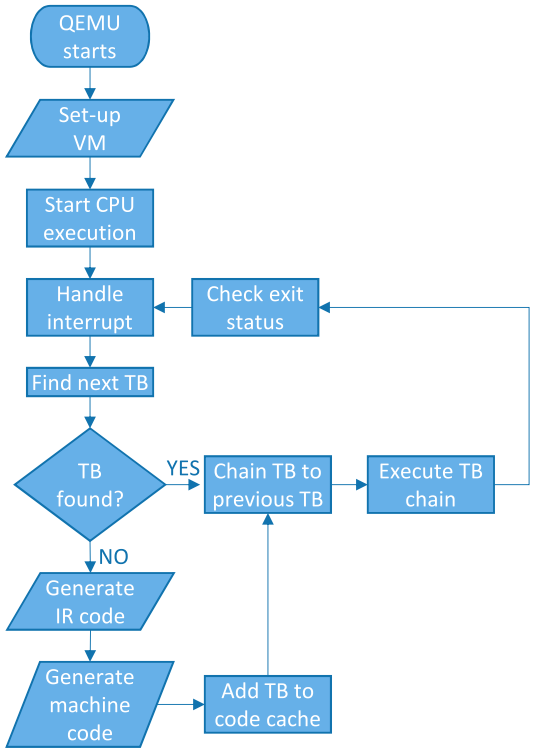


Figure 6.1.: QEMU's dynamic binary translation flow diagram.

Figure 6.1 illustrates in a flow diagram QEMU's run-time behaviour. Execution starts, and the first step is to set-up the Virtual Machine (VM) environment according to its specifications (e.g., number of CPUs, RAM size and available devices). Then, CPU execution starts with `cpu_exec()` function, referred to as the 'main execution loop'. Inside this execution loop, the first step is to handle the interrupts if any. Afterwards, `tb_find()` function searches the next TB according to the current PC value.

If no TB is found, the target machine code is generated through `tb_gen_code()` function, which subsequently call functions `gen_intermediate_code()` to translate source code into tiny code instructions and `tcg_gen_code()` to convert intermediate code into target machine code. After target machine code has been generated, the TB is stored in the code cache, `tb_jmp_cache`, at an index found by `tb_jmp_cache_hash_func()`. The generated/found TB is then chained to the previous TB, `tb_add_jump()`, to avoid a context switch in a following run. Finally, translated code execution continues through `cpu_loop_exec_tb()` function.

6.2.2 Rev.ng

Rev.ng is a binary analysis framework whose core element is, Revamb, its SBT tool, which combines the benefits of QEMU with those of LLVM. As described in Section 6.2.1, QEMU is a DBT based machine emulator, that follows a retargetable compiler like structure. Moreover, QEMU supports both system and user mode emulation. However, Rev.ng exploits just its front-end in user-mode to parse the source binary code and emit QEMU's IR. LLVM is a compilation framework that provides source and target independent optimisation support as well as resources for multiple machine code generation. The main components of LLVM's architecture are: (1) the front-ends, which translate source code in a variety of languages into LLVM IR. Clang, a C, C++ and Object-C front-end, is the one that has received the most attention; (2) Its IR, the core element in LLVM, a target-independent low-level programming language; (3) the Pass Framework, that is in charge of IR to IR transformation, most of the times seeking for code optimization and/or analysis; and (4) the back-end, which supports machine code generation for multiple instruction sets.

This tool suite currently supports static ARM, MIPS and x86-64 Linux binaries as input and can generate machine code for X86-64 output architecture. However, even if the current tool suite supports just a few input/output architecture combinations, the fact that it is based on QEMU and LLVM makes Rev.ng adaptable to other source/target architectures supported by QEMU [1] and LLVM [2] respectively.

As already mentioned, the core element in Rev.ng is its SBT tool. Revamb parses the statically linked Linux binary and uses QEMU's TCG as a front-end to generate tiny code instructions from any of the input architectures it supports. Then code in QEMU

[1] QEMU supports the emulation of various architectures including: Alpha, ARM, CRIS, x86, MicroBlaze, MIPS, OpenRISC, PowerPC, RISC-V, SH4, Sparc and their 64-bit variant when applicable.
[2] LLVM's back end supports many ISAs, including ARM, MIPS, PowerPC, Sparc, x86 and x86-64. However, just x86 (both 32-bit and 64-bit), ARM and PowerPC include most of the features.

IR form is further translated into LLVM IR instructions. However, in QEMU, certain features such as syscalls and complex instructions (e.g. floating point division) are handled through a set of external functions (written in C) known as helper functions. Therefore, using Clang, QEMU helper functions are obtained in the form of LLVM IR and statically linked before generating the LLVM module. Besides the helper functions, additional support is needed mainly for initialization purposes. To this end, revamb provides a set of support functions which are linked to the LLVM module. Then, the linked LLVM IR module is translated into machine code using LLVM compiler infrastructure. Figure 6.2 depicts the translation process of Rev.ng tool suite, which combines the use of QEMU's front-end and LLVM.

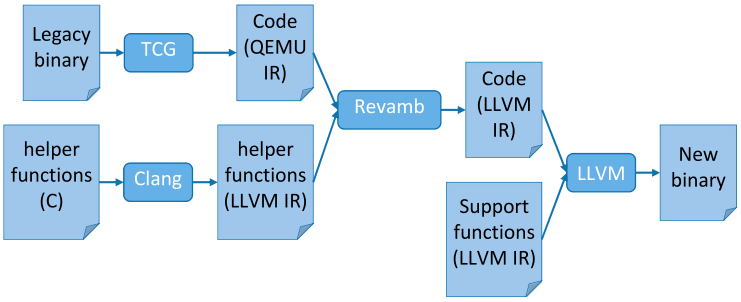


Figure 6.2.: Rev.ng's static binary translation process combining the use of QEMU, Revamb and LLVM.

6.3 Operating System – Linux

The use of Linux is a requirement implicit to both, the static and dynamic translation tools. Rev.ng translated a Linux binary from a set of source to a set of target ISAs. Therefore, the generated output binary is a Linux binary. Whereas QEMU DBT tool requires an OS to run, that can either be Linux, macOS or Windows.

Given that the described approach is desired to port code from a given embedded platform to another, for the implementation described in this chapter a minimal Linux distribution has been chosen. However, Linux is not a real-time OS and timing is one of the main concerns of this migration approach. Therefore, the PREEMPT_RT patch has been added to Linux kernel. The main purpose of this patch is to improve the RT behaviour on Linux by reducing the kernel's scheduler latency and response time. Moreover, PREEMPT_RT achieves a more deterministic Linux environment without the need for a specific API. To this end, the RT task running on top needs to be defined as so and given the highest allowed priority in the system [3].

[3] Given that PREEMPT_RT uses 99 as the priority for the kernel task sets and interrupt handler, the highest allowed priority is 98.

6.4 Timing Measurement and Control Blocks

Timing measurement and control blocks are a C++ extension, developed as a C++ library, that provides means for adding block-level timing annotations into embedded C/C++ applications. The blocks have been conceptually described in Sections 5.2.1 and 5.2.2, whereas this section describes their implementation.

6.4.1 Timing Measurement Block

The EET block, provides means to measure the end-to-end duration of a specific code section. The EET annotated application interacts with the block manager, timer and memory to implement the described behaviour. Figure 6.3 depicts the sequential diagram for the EET block. The application registers a new EET block in the block manager, which gets the current time instant. Then, after finishing execution of the wrapped code block, the block manager gets again the current time instant, computes the block's duration and saves the result in memory (under block's ID).

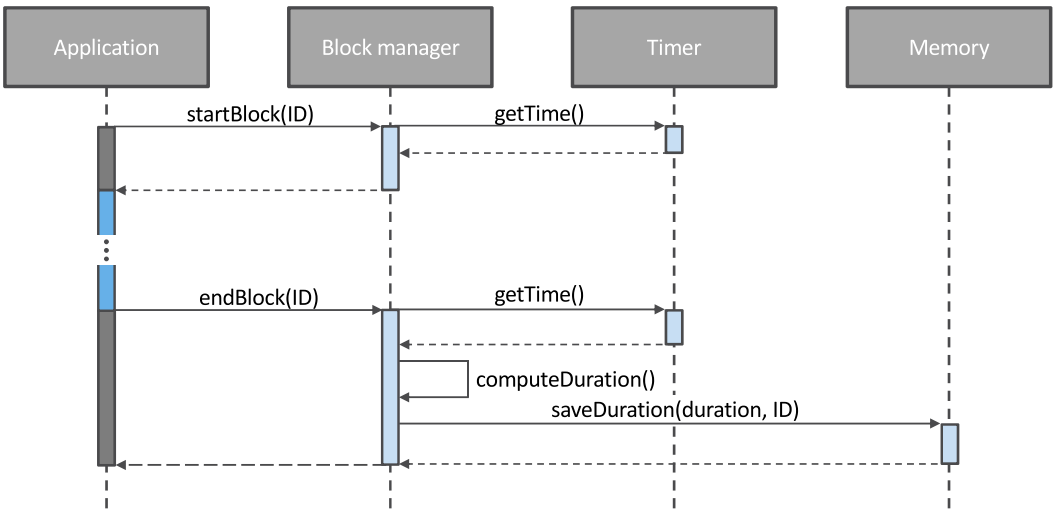


Figure 6.3.: EET block's execution – sequential diagram.

Within this research work, the EET block has a double use, it is used on top of the legacy hardware platform during the profiling phase and for timing analysis purposes, as well as on top of the new hardware platform for timing analysis purposes. Therefore, two variations of EET block are implemented; one considering a bare-metal development for the legacy hardware (ARM Cortex-A9), and the other for a Linux-based x86-64 architecture. Moreover, there is a third variant, which consists of a non-functional implementation. This particular EET block implementation is

used during the feasibility study of the dynamic binary translation approach (see Section 6.5.2 for further information).

Bare-metal EET block

From the multiple timers available on the Zynq-7000 (i.e., System Watchdog Timer, Triple Timer Counter, Global Timer Counter, CPU Watchdog, and CPU Private Timer), the Global Timer Counter has been chosen for the bare-metal implementation. The Global Timer is an automatically started, independently of the software, 64-bit auto-incrementing counter. As it is supported on Xilinx's Board Support Package (BSP), a collection of libraries and drivers that form the application's bottom layer, there is no need to do any particular adjustment on the hardware design for its implementation. Using the Global Timer, the duration of the EET block is computed and the elapsed time result is saved in memory for a latter analysis.

Linux-based EET block

The Linux version is implemented using the C++ based Chrono library to access a high-resolution clock and compute the elapsed time. Chrono library was designed to deal with the fact that clocks might differ across systems and clock and timer precision improves over time (see Chapter 5.7 in Josuttis [57]). A relevant aspect regarding Chrono is that it provides a precision-neutral solution by separating duration and time instants from specific clocks [28]. The elapsed time is then saved in a binary file for a latter analysis.

Structural EET block

The structural version of the EET block enclosures the annotated code section but does not implement the block's functionality. This block serves as a detection mechanism during the dynamic binary translation process. The adapted dynamic translator detects the structural EET block and implements its functionality (interacting with the block manager as well as host timer and memory).

6.4.2 Timing Control Blocks

Timing control blocks provide means to enforce a specific timing behaviour through a block-level source code annotation approach. They are used on the legacy hardware

platform for a midway test of the proposed timing enforcement solution during the migration process, as well as on the new hardware platform, where the dynamically or statically translated binary runs on top of Linux. Therefore, access to the timer within the timing control blocks is implemented using Chrono library. The library is used to access a high-resolution clock and assign real physical time to time points as well as to implement the time delay in PET and FET blocks. However, instead of using Chrono to operate over time points and time intervals with time units, time control blocks use Boost Units library, which provides means for time unit compile-time type checking as well as its conversion. Moreover, existing C++ language features have been used to wrap source code blocks and implement the corresponding time control functionality (according to the code wrapper type: PET, FET, BET or PNET) at a block scope level.

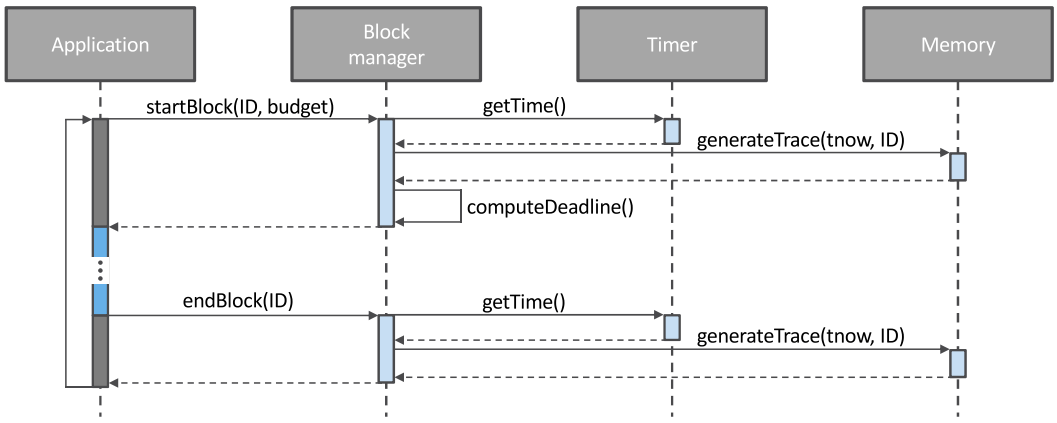
Time Control Blocks – Periodic Execution Time Block

The PET block annotated in the application interacts with the block manager, timer and memory to implement its behaviour as described in Section 5.2.2. Figures 6.4a, 6.4b and 6.5 depict the sequential diagram for the PET block in the three possible situations, block's execution: finished on time, finished earlier or exceeded the deadline.

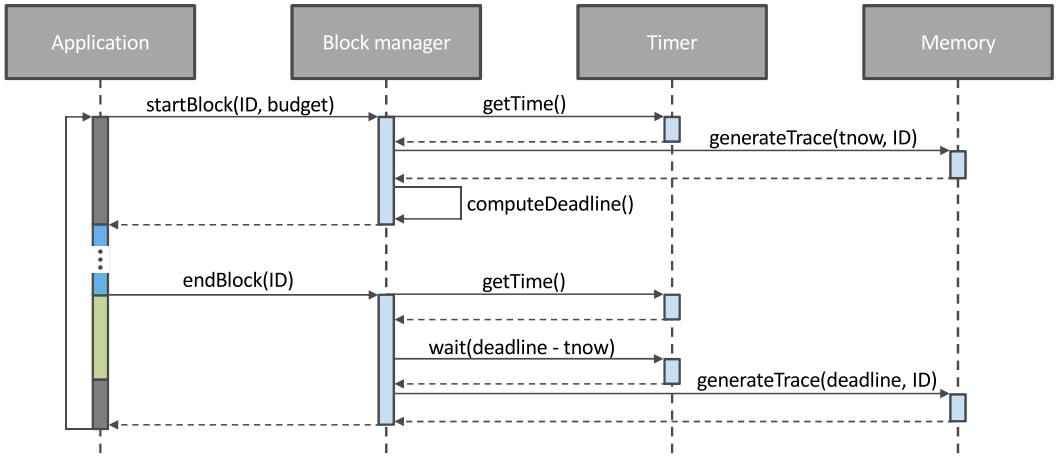
Sequence starts on the application, which registers the PET block in the block manager. The block manager gets the current time instant and generates a trace with the block ID and the corresponding time-stamp. Then, it computes the block's deadline according to the budget. After running the wrapped code block on the application, the block manager gets once again the time-stamp and generates the corresponding time trace. If the time-stamp matches the deadline (see Figure 6.4a), execution returns to the beginning of the application code. Whereas if the code block had finished before the deadline was met (see Figure 6.4b), the block manager will insert a delay to meet the deadline and execution will continue after the delay at the beginning of the application code. If on the contrary, the code block's execution exceeded the deadline (see Figure 6.5), an error handling routine will take place.

Time Control Blocks – Forced Execution Time Block

The FET block annotated in the application interacts with the block manager, timer and memory to implement its behaviour as described in Section 5.2.2. Figures 6.6 and 6.7 depict the sequence diagram for a FET block finishing on time and earlier than expected respectively. In the first case (Figure 6.6), the application registers



(a) PET block's execution on time



(b) PET block's execution early

Figure 6.4.: PET block's execution – sequential diagram for block's execution finishing on time and finishing early.

a new FET block in the block management object, which will get the current time instant. Then, the annotated code block runs until completion, that is when the bock management object will once again get the current time instant and compute the duration. As the block's duration matches that specified in the budget the application will continue execution with the next block. However, if as depicted in Figure 6.7 the block's execution finishes earlier than expected (actual duration is less than budget) a delay is inserted until the budget is met. Then, execution continues with the next bock. Whereas an error handling routine will take place, as shown in Figure 6.5, if the block runs longer than expected.

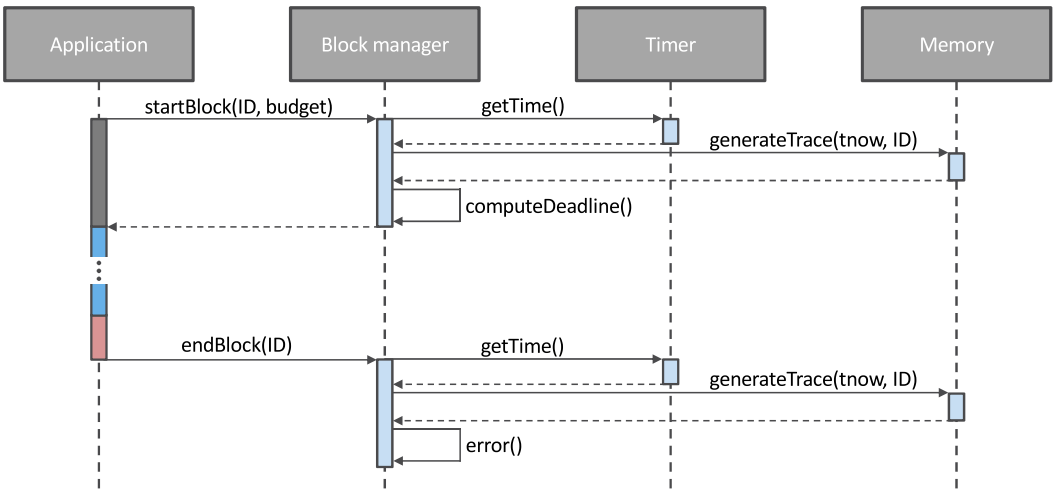


Figure 6.5.: PET block's execution late (same for FET and BET blocks) – sequential diagram for blocks' execution when budget is exceeded.

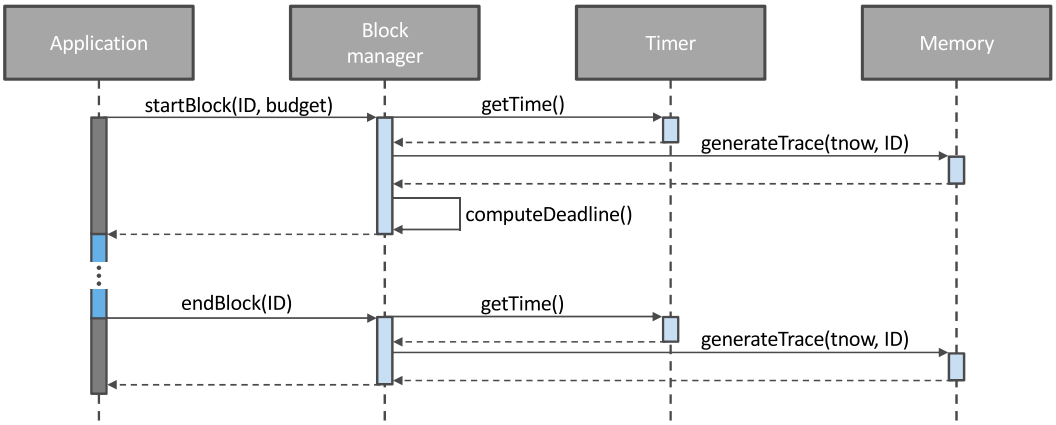


Figure 6.6.: FET block's execution on time (same for BET block) – sequential diagram for blocks' execution finishing on time.

Time Control Blocks – Budgeted Execution Time Block

The BET block can have three different behaviours depending on the wrapped code block's execution time. In order to deploy the block's functionality as described in Section 5.2.2, the application, block manager, timer and memory objects interact with each other. At the beginning of the BET block, the application registers the new block in the block manager, which will get the current time-stamp, generate the corresponding trace in memory and compute the block's deadline according to the budget. After running the wrapped application code, the block manager updates the time-stamp and accordingly generates the trace. If the deadline matches the time-stamp (see Figure 6.6), same as for the FET block execution will continue with the next block in the application code. Whereas if execution had finished earlier than

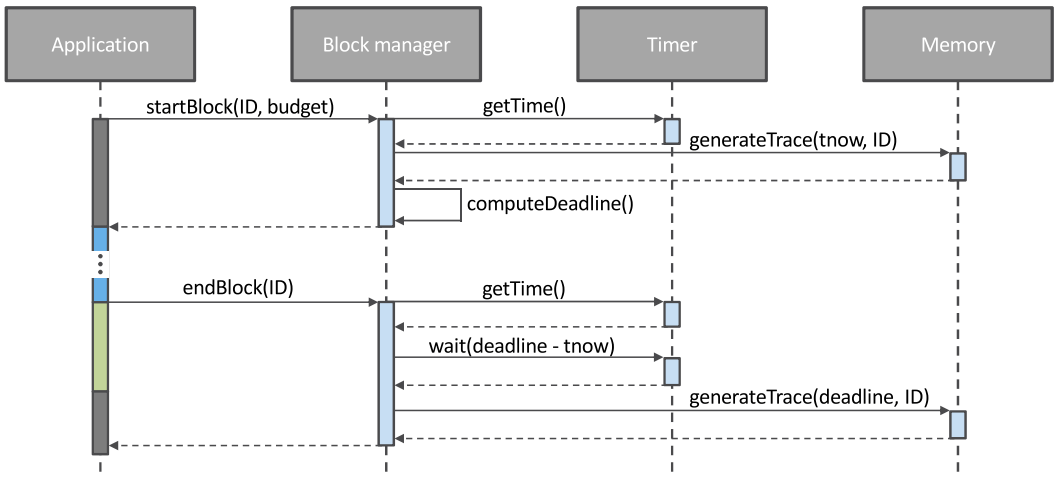


Figure 6.7.: FET block's execution early – sequential diagram for block's execution finishing early.

expected, as is the case in Figure 6.8, the block manager will compute the budget that will be passes to the next sibling BET or PNET block. In constrast, if execution took longer than expected, block's duration exceeded the budget (see Figure 6.5), as it is in PET and FET blocks, an error handling routine will take place.

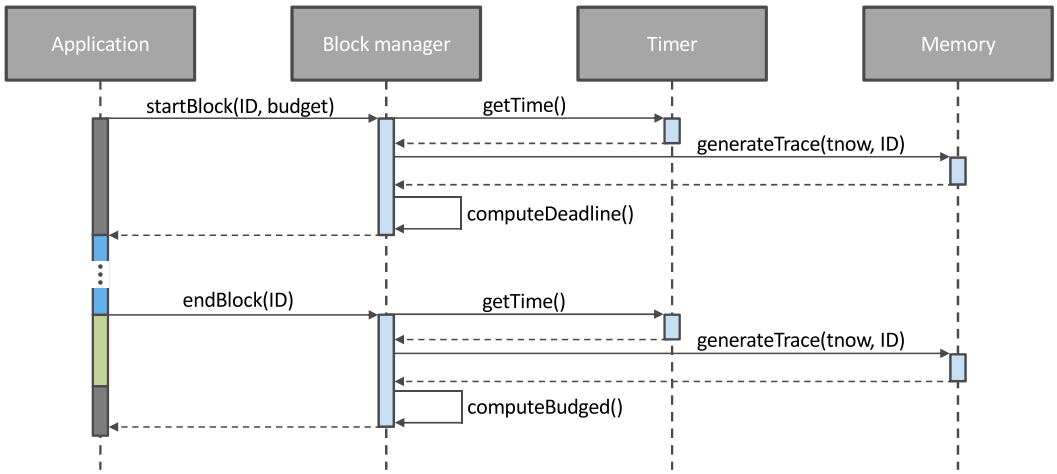


Figure 6.8.: BET block's execution early – sequential diagram for block's execution finishing early.

Time Control Blocks – Period N Execution Time Block

The PNET block can involve four different execution sequences according to execution's duration, but also according to the N^{th} period and offset arguments, since these arguments constraint the periods at which the wrapped code block is active (see Section 5.2.2 for more information on PNET block's functional behaviour).

Therefore, Figure 6.9 depicts the execution sequence when the block is not active. In that case, at the time of registering the new block, the block manager will realise that the block is not active and execution will continue with the next block in the application code.

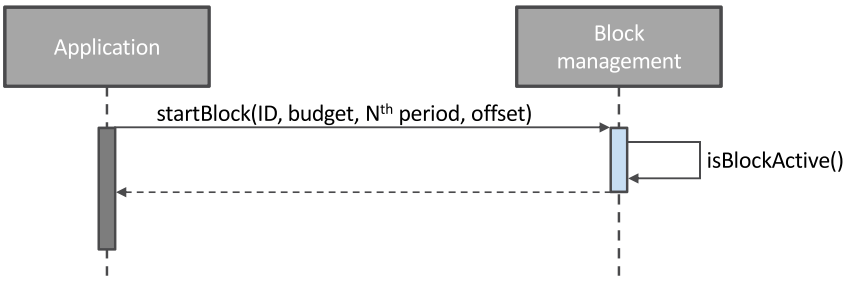


Figure 6.9.: PNET block execution not active – sequential diagram for block's execution when it is not active (according to N^{th} period and offset arguments).

If on the contrary, the PNET block is active on this period, possible execution sequences depend on block's execution finishing: on time, early or late. In the first case, depicted in Figure 6.10, the application object register the new block in the manager, which realizes the block is active and gets the time-stamp, generates the trace accordingly and computes the block's deadline according to the budget. After running the wrapped code, the block manger gets the new time-stamp and generates a new trace. As block's execution finished on time, the application will run the next block in the code.

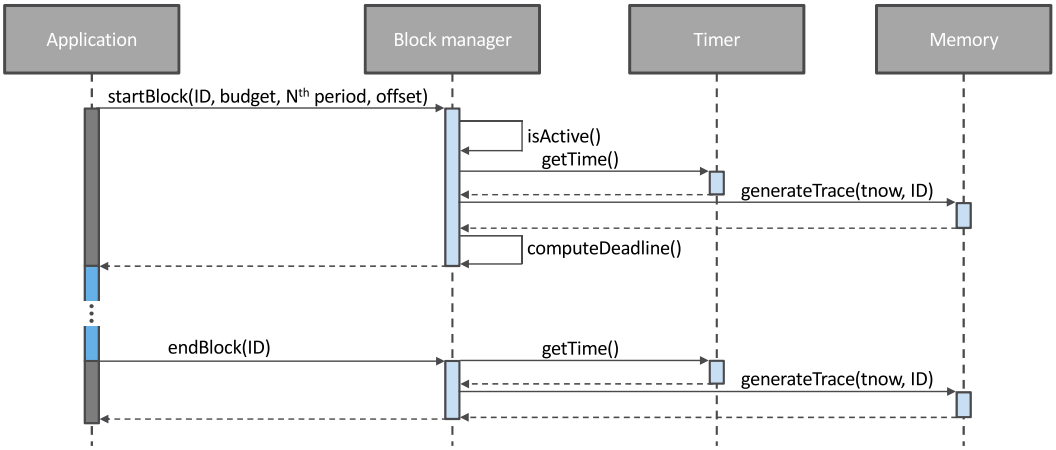


Figure 6.10.: PNET block's execution on time – sequential diagram for block's execution finishing on time.

In the case that the block runs faster than expected, as depicted in Figure 6.11, after running the wrapped code, updating the time-stamp and generating the new trace, the block manager will compute the budget that will be passed to the next sibling BET block. The execution will continue with the next block in the application. In

contrast, if the block runs slower than expected exceeding its deadline, as shown in Figure 6.12, instead of computing the budget, the block manager will run an error handling.

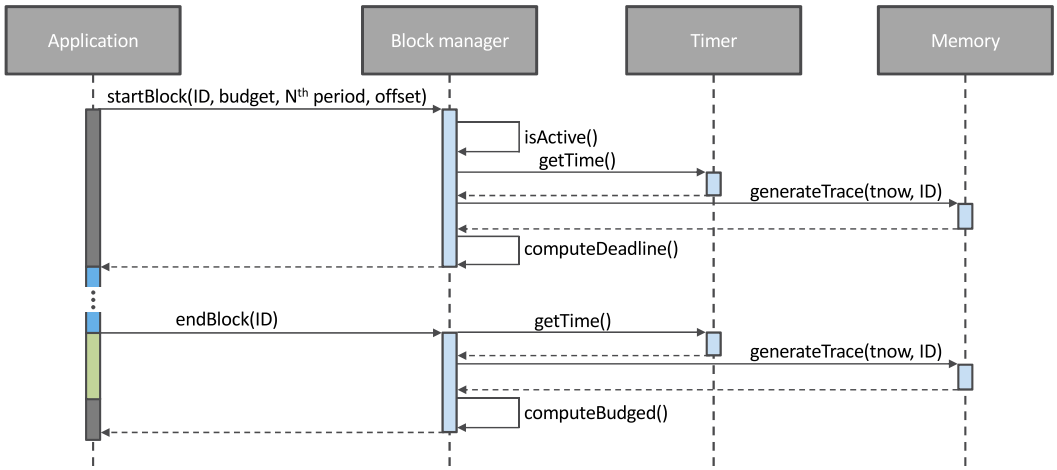


Figure 6.11.: PNET block's execution early – sequential diagram for block's execution finishing early.

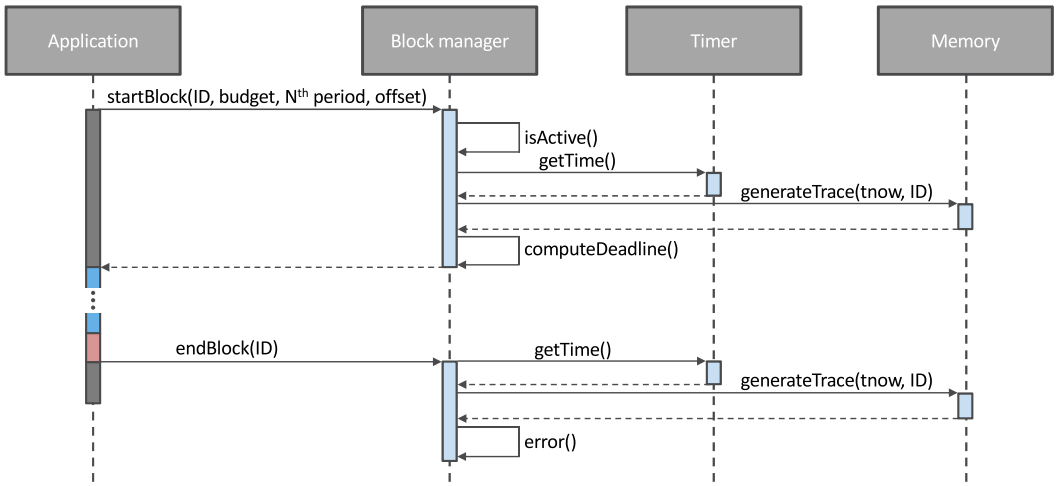


Figure 6.12.: PNET block's execution late – sequential diagram for block's execution when budget is exceeded.

6.5 Timing Measurement within Translated Binary

The timing measurement block is used in the legacy hardware platform to measure the end-to-end duration of legacy code. Moreover, it is integrated within the dynamic as well as the static binary translation tool in order to measure the end-to-end duration of the translated code. However, according to the chosen binary translation

approach, static or dynamic, annotations are handled in a different way. Section 5.4 describes the integration of timing blocks within the static/dynamic binary translation process in an implementation agnostic way. Whereas this section describes how the bare-metal implementation of the EET block is used on the legacy hardware platform and how QEMU and Rev.ng integrate the EET block (Linux-based and structural implementation) within their translation flow. This will provide means to measure the end-to-end duration of annotated code sections when running on the ARM Cortex-A9 and Intel Atom (after translation) processors, assessing the translation overhead with respect to timing as part of the feasibility study.

6.5.1 Legacy Platform – Timing Measurement

The legacy system follows the typical pattern of a reactive control system, as described in the system model presented in Section 5.1. The legacy application runs on the legacy hardware platform without any support of an operating system. Therefore, in order to measure the end-to-end duration of the legacy application, following Algorithms A.1.2 to A.1.4, the legacy code is systematically annotated with EET blocks. The annotated legacy code is then linked against the bare-metal implementation of the time measurement block and compiled for the legacy hardware platform. The annotated legacy binary runs on top of the legacy hardware platform and the required timing-related data is collected. Figure 6.13 shows on the left side the linking and compilation of the annotated legacy code. Whereas on the right side the runtime architecture of the annotated legacy binary is shown, where the annotated legacy binary runs on top of the ARM Cortex-A9 processor.

6.5.2 Dynamic Approach – Timing Measurement

The dynamic approach takes advantage of QEMU [16], a machine emulator built upon a portable and low overhead DBT tool, to translate legacy code at run-time. As timer virtualization is not supported, in order to access the target timer and implement timing measurement block's functionality, apart from annotating the legacy behavioural code, QEMU's source code has to be adapted before translation. Then, the annotated legacy code is linked against the structural TMCB library and compiled for the legacy processor, whereas the adapted QEMU is linked against the Linux version of the TMCB library and compiled for the new processor. The annotated legacy binary runs on top of the adapted QEMU, which runs on top of a minimal Linux distribution configured using the PREEMPT_RT patch. Moreover,

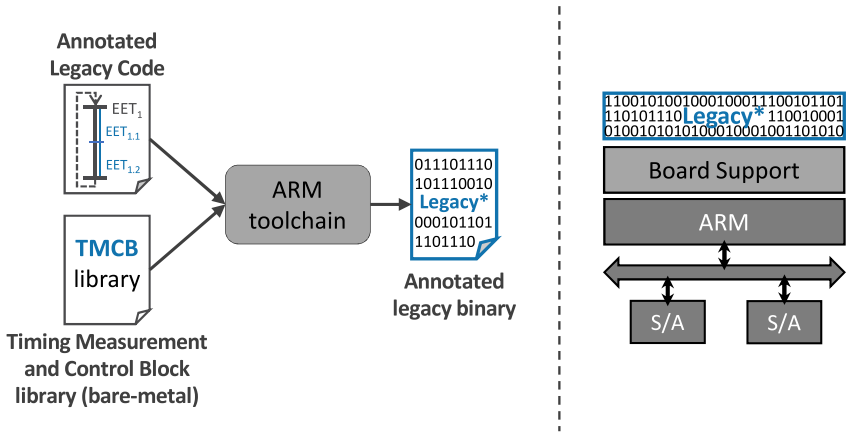


Figure 6.13.: On the left side the linking and compilation process of the EET annotated legacy code is shown. The annotated legacy code is linked against the bare-metal implementation of the Timing Measurement and Control Block (TMCB) library and compiled using the ARM toolchain. On the right side the runtime architecture is shown, where the annotated legacy binary runs on top of the ARM Cortex-A9 processor.

when launching QEMU, the `chrt` command is used to manipulate the real-time attributes of the process and run it with the highest allowed priority. Figure 6.14 depicts on the left side the linking and compilation process of the annotated legacy code and adapted DBT tool, whereas the right side of the figure shows the runtime architecture, which includes the dynamic translation process. In the following the annotation of legacy code and adaptation of QEMU are covered.

Annotate Legacy Code

Through Algorithms A.1.2 to A.1.4, the legacy code is systematically annotated with EET blocks. The annotated code is then linked against the TMCB library, where the EET block is implemented following the structural version described in Section 6.4.1. The annotation point to be detected by QEMU is a function call, this way we ensure that there is a branch in the code, consequently this instruction will be the first ones in its corresponding TB.

Adapt QEMU

QEMU's source code has to be adapted in such a way that it identifies the structural EET block annotations in the legacy application and implements their functionality. Figure 6.15 illustrates in a flow diagram the run-time behaviour of the adapted QEMU, which is based on the original translation flow presented in Figure 6.1.

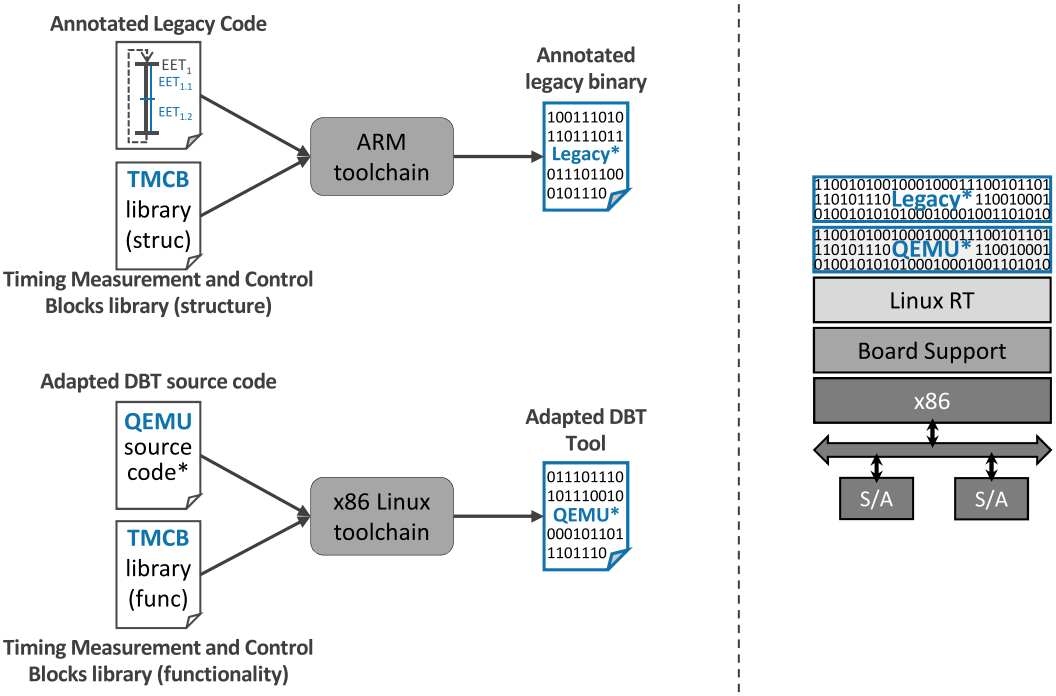


Figure 6.14.: On the left side the linking and compilation process of the EET annotated legacy code and adapted DBT tool are depicted. The annotated legacy code is linked against the structural implementation of the TMCB library and compiled using the ARM toolchain. The adapted DBT source code is linked against the Linux implementation of the TMCB library and compiled using the x86 Linux toolchain. On the right side the runtime architecture is shown, which includes the dynamic translation process.

QEMU identifies TBs according to the PC value of the first instruction in the block. Since the annotation points consist of function calls, it is ensured that the annotations are the first instruction in the TB. So, if the generated/found TB corresponds to the annotation's PC value an auxiliary code is inserted that, through the Linux implementation of the EET block, gets the start/end time, measures the end-to-end duration, and saves obtained data in memory for the latter analysis phase. However, as QEMU chains consequent TBs, start and end TBs would be chained to previous TBs and it would not be possible to detect them. So, it is necessary to ensure that the TBs containing the annotation are never chained to the previous one.

6.5.3 Static Approach – Timing Measurement

The static legacy code migration approach employs a suite of tools for binary analysis, Rev.ng [40], to translate a statically linked Linux binary into an equivalent target machine code. In order to measure the end-to-end duration of the translated code,

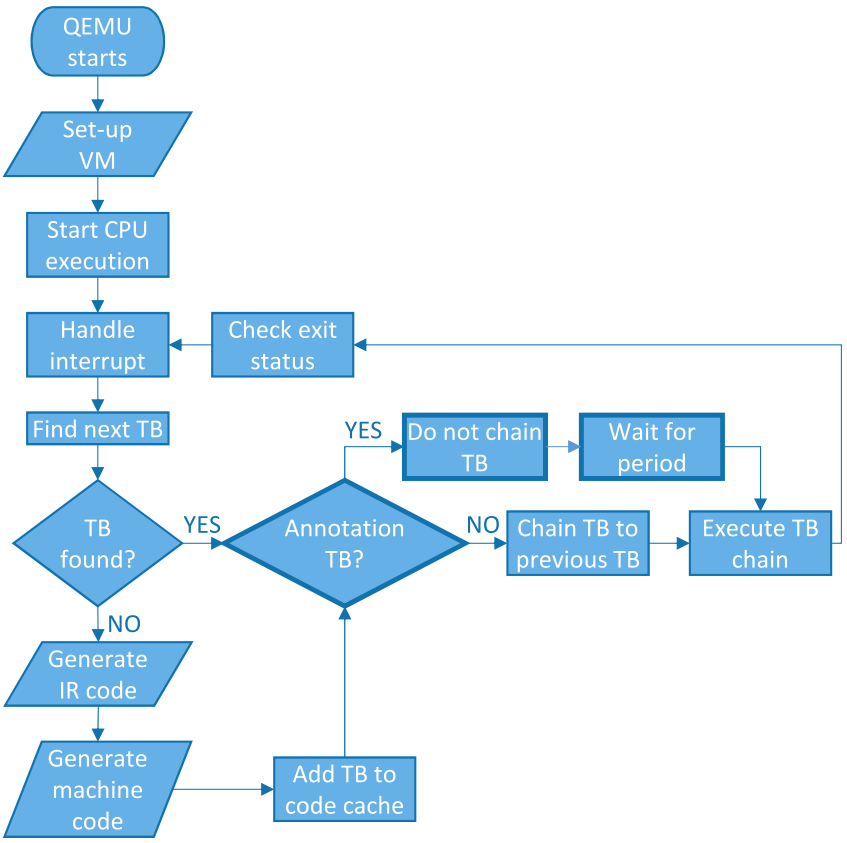


Figure 6.15.: QEMU's dynamic binary translation flow diagram adapted to detect EET block's and insert their functionality.

following Algorithms A.1.2 to A.1.4, the legacy code is annotated with EET blocks. Using a Linux toolchain for the legacy architecture, the annotated code is then statically linked against the TMCB library, where the EET block is implemented following the Linux version, as described in Section 6.4.1 and compiled for the legacy architecture. The statically linked binary is then translated off-line, before execution (see left side of Figure 6.16. Once translated, the new binary runs on top of a minimal Linux distribution that has been configured with the PREEMPT_RT patch. When launching the translated binary, the `chrt` command is used to run the application with the highest allowed priority. On the right hand of Figure 6.16 the run-time architecture of the static migration approach, as it has been described, is shown.

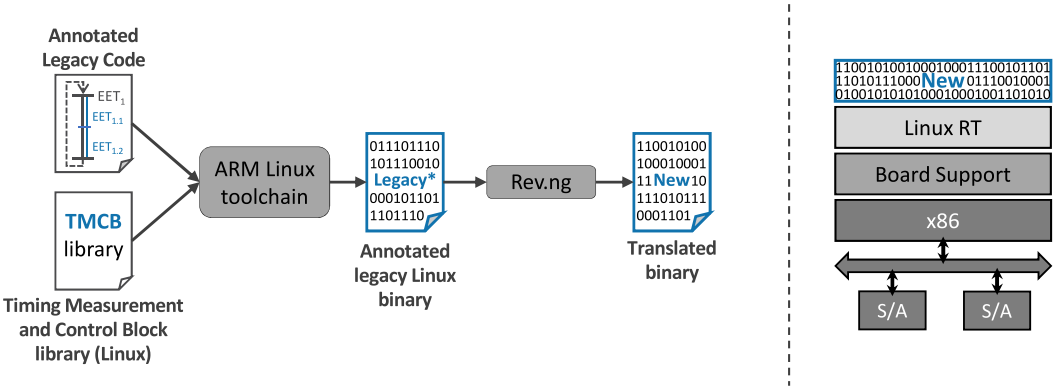


Figure 6.16.: On the left side the linking, compilation and static translation process of the annotated legacy code is depicted. The annotated legacy code is linked against the Linux implementation of the TMCB library and compiled using the ARM Linux toolchain. The annotated legacy Linux binary is statically translated, using Rev.ng, to obtain the translated binary. On the right hand the runtime architecture is shown, where the translated binary runs on top of Linux RT on the new architecture (x86).

6.6 Timing Control within Static Binary Translation

Integrating the timing enforcement described in previous section (see Section 6.4.2) within the SBT process, a timing-aware translation is achieved. As already stated in previous sections, the static approach takes advantage of the Rev.ng binary analysis tool-suite to statically migrate the legacy code. Through the profiling phase (see Section 5.2.1), the behavioural legacy code and legacy timing properties are extracted. Then, the behavioural legacy code is systematically annotated with timing control blocks in order to make legacy timing properties explicit. The systematic annotation of timing control blocks is covered through Algorithms A.2.1 to A.2.4 where the legacy system's model is transformed into a time control block based annotated legacy code. As it is done with time measurement block based annotated legacy code, the time control block based annotated behavioural legacy code is statically linked to the Linux version of the TMCB library and compiled using the ARM Linux toolchain. The technical implementation of TMCBs as a library enables its use across multiple platforms. Then, the annotated binary is statically translated from ARM to x86 ISA using Rev.ng tool-suite. The translated binary runs on top of a minimal Preempt-RT Linux distribution on the new Intel Atom processor.

6.7 Testing Timing & Functional Properties within Migration Flow

Testing of timing as well as testing functional properties is carried out several times within the RT legacy software migration flow (see Section 5.3.1, steps 2a and 2b in Figure 5.1). Sections 5.3.1 and 5.3.2 describe, respectively, the testing of timing and functional properties from an implementation agnostic point of view, whereas this subsection sets the focus on the implementation of the test framework for such testing process.

6.7.1 Testing Timing Properties

To validate the timing behaviour, the systematically (through Algorithms A.2.1 to A.2.4) annotated time control blocks generated time traces at runtime, which together with systematically (through Algorithms A.3.1 to A.3.5) obtained formal timing specifications and MULTIC tool provide means to validated the timing behaviour. The first timing validation comes after the lifting of timing properties. The annotated behavioural legacy code, that has been linked against the Linux version of the TMCB library and compiled for the ARMv7-A ISA, runs on top of a minimal Preempt-RT Linux distribution on the legacy ARM Cortex-A9 processor to obtain the corresponding time traces, which are then validated. Timing is validated once again after the static translation of the annotated legacy binary. Using Rev.ng, the annotated binary for the ARMv7-A ISA is translated to a equivalent (from the functional and timing behaviour perspective) binary for the x86 ISA. The new binary, which integrates the timing enforcement, runs on top of the Intel Atom E3866 processor and the corresponding time traces are obtained for the latter validation.

6.7.2 Testing Functional Properties

Functional behaviour validation is carried out using a set of reference I/Os, which are obtained for the legacy applications running on the ARM Cortex-A9 processor. After the lifting of timing properties the functional behaviour is validated for the first time. The systematically (following Algorithms A.2.1 to A.2.4) annotated behavioural legacy code, that has been linked against the Linux version of the TMCB library and compiled for the ARMv7-A ISA, runs on top of the ARM Cortex-A9 processor. On runtime, the annotated application is feed with the reference input data and obtained

output data is collected. After execution, collected output data is compared to the reference output data to validate the functional behaviour. Then, the annotated ARM binary is statically translated into an equivalent (from the functional and timing behaviour perspective) binary for the x86 ISA. The new binary runs on the Intel Atom E3866 processor and is feed with the reference input data to collect the corresponding output data, which is compared to the reference output data after execution. This way the functional behaviour is validated after the timing-aware translation process.

7 Evaluation Process and Result Analysis

The following sections evaluate the RT legacy software migration flow described in Chapter 5. The first section presents an overview of the following sections, mapping each step on the evaluation process to the contributions that are evaluated and the answered research questions. Section 7.2 introduces the WCET benchmarks and applications that take place in the evaluation process. Then, each of the evaluation steps is presented in a dedicated section. Section 7.3 described the feasibility study of two machine-adaptable binary translation tools for their use in a timing property conserving migration process. Then Section 7.4 describes the evaluation of the implement block level timing enforcement mechanism as well as the systematic annotation process. Finally Section 7.5 evaluates the timing-aware legacy software migration approach.

7.1 Overview & Organization

As already stated, the evaluation process consists of three main steps: a feasibility analysis, the timing enforcement assessment and the timing-aware static migration assessment. Figure 7.1 depicts a summary of each of the steps in the evaluation process.

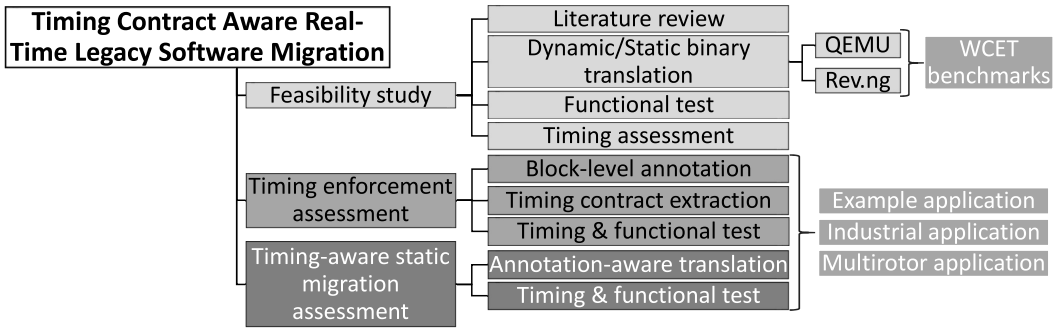


Figure 7.1.: Evaluation overview and organization.

The feasibility study itself is the first contribution (***C1***) of this work. To this end, through the analysis of the SotA two machine-adaptable binary translators are selected and their feasibility to port RT legacy software is analysed. This evaluation step gives an answer to the first and second research questions (***RQ1*** and ***RQ2***). The related work analysis serves to find out how legacy code can be ported to a new architecture while preserving its timing as well as functional behaviour. Then, the selected machine-adaptable static and dynamic binary translators are assessed with respect to timing to determine the constraints of binary translation techniques when porting RT legacy software.

The timing enforcement assessment evaluates the block-level timing annotation mechanism. The temporal constructs developed as part of Contribution ***C2*** are used to systematically annotate legacy timing properties into the behavioural legacy code in Contribution ***C3***. Moreover, to evaluate the timing behaviour on the annotated code Contribution ***C4*** systematically transforms legacy timing properties into formal timing specifications, which are used together with the time traces generated at runtime by the temporal construct to perform the timing test on MULTIC tool. This process answers to ***RQ3*** describing how to express and annotate into the legacy application expert knowledge related to legacy timing, as well as evaluating the proposed timing annotation mechanism.

Finally, the third step in the process evaluates Contribution ***C5***, the integration of the timing enforcement mechanism into the binary translation process. To this end, the formal timing specification of Contribution ***C4*** are needed. As a result of this evaluation step an answer is given to Research Questions ***RQ4*** and ***RQ5***. The technical implementation of the temporal constructs is presented as a possible solutions to preserve the annotated timing behaviour during the migration process. Whereas, the formal timing specifications and time traces generated at runtime by the temporal constructs serve, together with MULTIC tool, on the validation of the timing behaviour on the new architecture. As the final step the migration flow is being analysed, this step serves also as an evaluation of the whole migration process described in Chapter 5.

7.2 Evaluation Software

Multiple benchmarks and applications take place in the evaluation process, starting from the Mälardalen WCET benchmarks and followed by the example, industrial and multirotor applications. The WCET representative benchmarks contain a great variety of algorithms (including loops, nested loops, use of array and/or matrices,

and use of floating point operations), therefore, their use in the feasibility study ensures a wide analysis of the timing behaviour of the proposed static and dynamic solutions. For the timing enforcement and timing-aware migration assessment the three applications have been used. The example application is a complete application in the sense that it contains tasks with different periods and some of the tasks are considered critical code sections and hence through the example application every timing control block can be evaluated, analysing this way corner cases. However, this is a self generated application and therefore the great effort of porting non-proprietary code is excluded. On the contrary, the industrial application is real legacy industrial code that was implemented by a third party. For this reason, it is a suitable application to evaluate the effort of porting non-proprietary legacy industrial code. The main drawback of using the industrial application on the evaluation is its confidentiality, which hinders the open discussion of obtained results. Moreover, through the industrial application not every corner case is analysed. To cover the mentioned drawback, the multirotor application is evaluated. This is also not self implemented legacy code and hence serves to evaluate the effort of porting third party code with the advantage of being public, which means that results can be freely discussed. As for the industrial application, every corner cases is not covered, but the analysis of corner cases has been achieved through the example application.

7.2.1 Mälardalen WCET benchmarks

The Mälardalen WCET research group provides a suite of WCET representative benchmark programs to make comparable WCET estimates derived by WCET analysis tools. Given the great amount of benchmarking suites for computer science, this benchmark suite collects those which have most relevance to WCETa analysis. In the following the Mälardalen WCET benchmarks are classified for the latter timing assessment as part of the feasibility study (see Section 7.3).

Classification

When analysing the results, benchmarks are classified into short- and long-running according to their average execution time on the legacy hardware. We consider a benchmark to be short-running bellow 100000 ns and long-running over 100000 ns (measured on the legacy processor). Moreover, for a better result analysis, benchmarks are classified according to their characteristics (see Table 7.1). To the information provided by the Mälardalen WCET research group [46], we have added a column to classify the benchmarks depending on whether they are dominated by (1)

complex computations, (2) simple computations or (3) control flow statements. This classification is closely related to the next three columns, where branching, memory, integer and floating point utilization has been classified as high (H), medium (M) or low (L). The first group is expected to have a low translation overhead, since the new processor can handle better complex computations. The second group also, since the translator can efficiently translate this code. Whereas the third group is expected to have a high translation overhead, since control flow statements hinder translation efficiency, mainly in the dynamic approach due to the difficulties to apply TB-chaining, but also in the static approach because statements might depend on run-time behaviour.

Table 7.1.: Benchmark classification. **S** = always single path program. **L** = contains loops. **N** = contains nested loops. **A** = uses arrays and/or matrixes. **B** = uses bit operations. **R** = contains recursion. **U** = contains unstructured code. **F** = uses floating point calculation. **CC** = code dominated by complex computations. **SC** = code dominated by simple computations. **CF** = code dominated by control flow statements.

Benchmark	S	L	N	A	B	R	U	F	CC/SC/CF	Branching	Memory	Integer	Floating Point
adpcm	-	✓	-	-	-	-	-	-	CF	H	L	H	-
bs	-	✓	-	✓	-	-	-	-	CF	H	M	L	-
cnt	-	✓	✓	✓	-	-	-	-	CF	H	M	L	-
compress	-	✓	✓	✓	-	-	-	-	CF	H	M	M	-
cover	✓	✓	-	-	-	-	-	-	CF	H	L	L	-
crc	✓	✓	-	✓	✓	-	-	-	CC	M	M	M	-
duff	✓	✓	-	-	-	-	✓	-	CF	H	M	L	-
edn	✓	✓	✓	✓	✓	-	-	-	CC	M	H	H	-
expint	✓	✓	✓	-	-	-	-	-	SC	M	L	M	-
fac	✓	✓	-	-	-	✓	-	-	CF	H	L	M	-
fdct	✓	✓	-	✓	✓	-	-	-	CC	L	H	H	-
fft1	✓	✓	✓	✓	-	-	-	✓	CC	M	H	L	H
fibcall	✓	✓	-	-	-	-	-	-	CF	M	L	L	-
fir	-	✓	✓	✓	-	-	-	-	SC	M	M	L	-
insertsort	-	✓	✓	✓	-	-	-	-	SC	M	M	L	-
janne_complex	✓	✓	✓	-	-	-	-	-	CF	H	L	L	-
jfdctint	✓	✓	-	✓	-	-	-	-	SC	L	H	H	-
lcdnum	-	✓	-	-	✓	-	-	-	CF	H	L	L	-
lms	✓	✓	-	✓	-	-	-	✓	CC	H	L	L	H
ludcmp	-	✓	✓	✓	-	-	-	✓	CC	H	H	L	H
matmult	✓	✓	✓	✓	-	-	-	-	SC	M	H	M	-
minver	✓	✓	✓	✓	-	-	-	✓	CF	H	H	L	M
ndes	-	✓	-	✓	✓	-	-	-	CC	L	M	H	-
ns	-	✓	✓	✓	-	-	-	-	CF	H	H	L	-
prime	✓	✓	-	-	-	-	-	-	SC	M	M	L	-
qsort-exam	-	✓	✓	✓	-	-	-	✓	CF	H	H	L	M
qurt	✓	✓	-	✓	-	-	-	✓	CC	M	H	L	H
recursion	✓	-	-	-	-	✓	-	-	CF	H	L	L	-
select	-	✓	✓	✓	-	-	-	✓	CF	H	H	L	M
sqrt	✓	✓	-	-	-	-	-	✓	CC	H	L	L	H
st	✓	-	✓	-	✓	-	-	✓	CC	M	M	L	H
statemate	-	✓	-	-	-	-	-	-	CF	H	L	L	-

7.2.2 Example application

The example application, first introduced in Section 5.1.3, resembles the typical pattern of a reactive control system. This bare-metal application includes seven periodic tasks (with different periods) executed following a static scheduling policy. The functionality of the example application consists on a bit toggling sequence, where each of the sequential tasks toggles a specific bit in an output variable (the bit in the position corresponding to the task number, the bit in position zero never toggles).

7.2.3 Industrial application

The industrial application is also a reactive control application consisting of a periodic task that runs every 1 ms. This application is an industrial legacy application that belongs to an industrial field client from IKERLAN and it is therefore confidential. For this reason, the functional behaviour of this application cannot be described. Beside the inconveniences derived from using a confidential application in the evaluation process, the interest on porting this application lies in the intention to prove the applicability of the approach to a real legacy industrial applications.

7.2.4 Multirotor application

Multirotors are remotely piloted air-crafts with multiple applications fields such as agriculture [1], construction [2], aerial photography, fire-fighting [3], and retail and delivery [4] among many others.

The multirotor application that will be used in the evaluation process is the OFFIS multirotor [77], which supports a mixed-critical architecture and enables high-performance processing while still supporting a safe flight control. As part of the public demonstrator in SAFEPOWER project [38], OFFIS multirotor was extended to support low-power management methods to reduce power consumption without jeopardizing the safety critical system's safety requirements.

[1] DroneSeed developed drones for insecticide spreading, and is developing drones able to plat trees
[2] ETH Zürich is working in an aerial construction research project
[3] DroneFly has developed drones, equipped with thermal cameras, for monitoring as well as individual search and rescue purpose
[4] Amazon is working in a the Prime Air delivery project

The flight control algorithm is responsible for computing the motor values (control variable) based on the control orders from the user (set-point) and the sensor data (process variable). Figure 7.2 depicts the main components of the flight controller, which are *Read Sensors*, *Sensor Processing*, *Read Remote*, *Remote Processing*, *Flight Controller*, and *Send Motor Values*. To guarantee a stable flight behaviour, an update rate of 500 Hz must be ensured in the motor drivers, therefore the control cycle cannot exceed 2.1 ms.

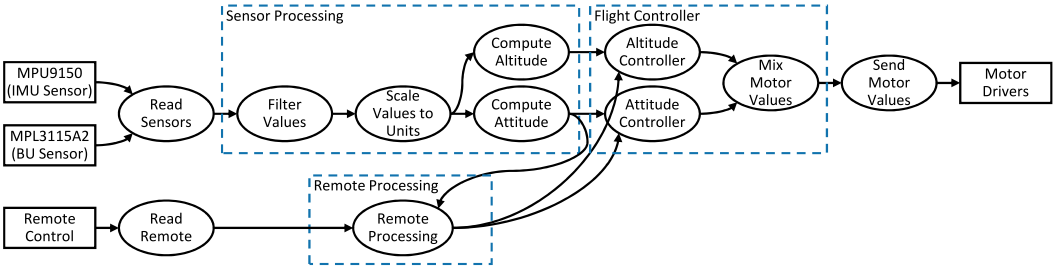


Figure 7.2.: Overview of the flight controller [82]

7.3 Feasibility Study – Dynamic vs. Static Binary Translation

The feasibility study starts with the selection of the dynamic and static binary translation tools under test. Therefore, the first subsection reasons about this selection process. The evaluation set-up is described then, followed by the corresponding result's analysis, which consists of the translation overhead analysis and the static vs. dynamic migration comparison. Finally from the result analysis the obtained conclusions are presented.

7.3.1 Translation tool selection

According to the literature review in the are of cross-platform machine-adaptable binary translators (see Section 3.2), two binary translation tools, one dynamic and the other static, have been selected for the feasibility study. As already stated in Section 6.2, the selected dynamic and static tools are QEMU and Rev.ng respectively.

Several aspects, relevant when targeting a timing-aware legacy software migration approach, have been considered for the selection of these tools. On the one hand, a cross-platform translator is as essential requirement given the main objective of this research work. On the other hand, a machine-adaptable binary translator is also of

great interest. Since for the fact that BT tools are highly dependent on source and target architectures, developing a BT tool from scratch entails a great effort. QEMU as well as Rev.ng comply with these aspects, offering a cross-platform source and target machine-adaptable binary translator. Moreover, they are both open-source solutions.

7.3.2 Evaluation set-up

In order to perform a feasibility analysis of the static and dynamic binary translation tools for they use in a timing property conserving migration process, a test framework has been constructed. This framework provides means to perform a measurement based WCET analysis on the legacy platform and on the new hardware platform using both migration approaches, static and dynamic. The feasibility test measures the execution time of a selection of WCET representative benchmark programs, provided by the Mälardalen WCET research group [46]. Together with the WCET representative benchmarks, the execution time of an empty application [5] has also been analysed. Measuring the empty application execution time provided a mean to measure the overhead introduced by the underlying system on either migration approach, which is composed of: QEMU and Linux PREEMPT_RT on the dynamic solution and the extra instructions inserted on the code by Rev.ng translator and Linux PREEMPT_RT on the static solution. The obtained results are then analysed and compared in Sections 7.3.3 and 7.3.4.

As already stated in Section 6.1, the test framework has been implemented on top of the following two Evaluation Boards (EBs): the ZC702 with a Zynq-7000 XC7Z020 SoC (consisting of a FPGA and an ARM Cortex-A9 processor with an operation frequency of 666 MHz) and the MinnowBoard Turbot Dual-Core with a Dual-core Intel Atom E3826 processor with an operation frequency of 1463 MHz. The former is employed as the source processor (legacy), whereas the latter is used as the target processor where the static and dynamic legacy code migration techniques are tested [6]. To perform the execution time measurements on the ARM processor the bare-metal implementation of the timing measurement block has been used, as described in Section 6.5.1. Whereas to measure the execution time on the Intel Atom the Linux-based implementation and the structural implementation of the timing measurement block have been used. Sections 6.5.2 and 6.5.3 describe

[5] We consider an empty application that whose main function does not contain any instruction.

[6] Despite the fact that the Cortex-A9 processor is not a legacy hardware platform (which is usually slower than the new hardware platform), it has been chosen for the feasibility analysis for the fact that it is supported by the selected SBT tool. However, Rev.ng can be "inexpensively" adapted to support other source/target ISAs.

how the measurements are done when applying the dynamic and static approach respectively. The annotated benchmarks and empty application are compiled without any optimization. This is a common industrial practice, since usual compilers have no integrated notion of timing and applied optimizations may lead to large WCET degradations [39].

To get the results, each annotated benchmark and the annotated empty application run 15000 times (statistically representative enough) on the ARM Cortex-A9 and Intel Atom E3826 processors. Through the annotated EET blocks results are saved in memory and analysed afterwords. Then, the feasibility survey compares the execution time of the Mälardalen WCET benchmarks and the empty application running on top of the legacy hardware platform and the new hardware platform using both, dynamic and static migration approaches. Given that we are targeting the migration of a reactive control system where the application is periodically triggered, the first runs can be excluded from the analysis. This way the DBT warm-up time, code translation/optimization overhead on the first runs when there is still no translated code available in the code cache, does not affect the measurements. Moreover, to avoid the overhead of launching QEMU itself, the benchmarks are periodically executed without re-launching QEMU.

7.3.3 Translation overhead analysis

The translation overhead analysis is performed based on the empty application, measuring the execution time on the legacy and new ISAs following a dynamic and static translation process. Figure 7.3 shows the collected data distribution with a zoom in the maximum execution time result area. Table 7.2 contains the minimum, maximum, average, standard deviation and 99%-quantile of the collected data.

Results show that, as expected, the average translation overhead of the dynamic binary translation solution is higher than the average static approach overhead, almost 3.7x greater. In the dynamic approach, translation and optimization counts on the measured execution time and even though QEMU applies counter measures, such as translated code caching and consequent TB chaining, it still implies great overhead. Whereas the static approach is capable of generating more efficient code, since neither translation nor optimization counts on the execution time. Therefore, it is possible to apply more aggressive optimizations.

Regarding the 99%-quantile, which indicates the value bellow which the 99% of the measured values are found, the difference between the static and dynamic migration approaches is even greater. The 99%-quantile in the dynamic migration approach is

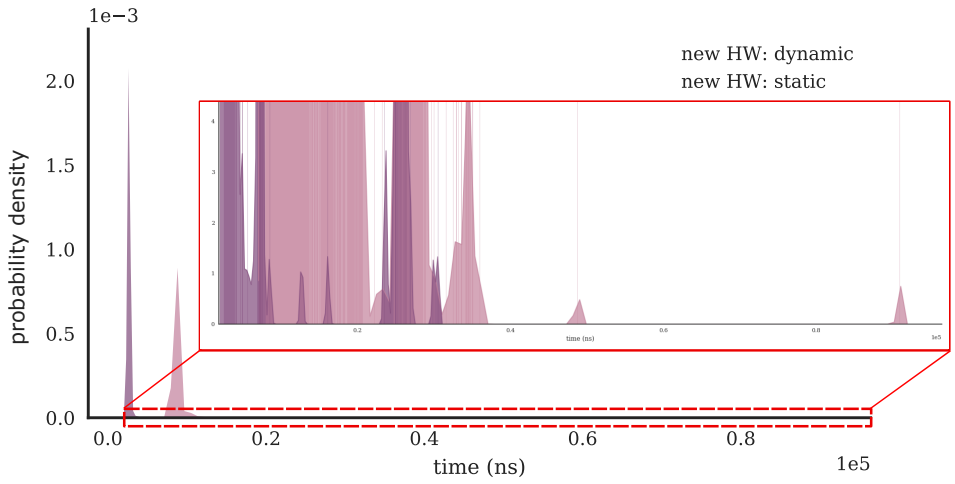


Figure 7.3.: Distribution of execution time data collected running the empty application on the Intel Atom E3866 processor following a dynamic/static translation process. The x-axis and the y-axis are respectively the measured duration for each execution of the empty application and the frequency of occurrence of each measurement. In the zoom-in area it can be appreciated that the dynamic approach (light pink) has sporadic corner execution time values that differ greatly from the average.

13x higher than that in the static approach and 3.9x higher than the dynamic average execution time. Whereas the 99%-quantile in the static approach is just 1.1x higher than the static average execution time. The standard deviation is similar in both migration approaches, 58.9% of the average in the dynamic vs. 51.5% in the static approach. These results lead to the conclusion that although both approaches have little difference on the maximum execution time, these sporadic corner execution time values, which can be appreciated in the zoom-in area in Figure 7.3, are more

Table 7.2.: Maximum, minimum, average, standard deviation and 99%-quantile of the measured execution time when running the empty application on the Intel Atom E3866 processor following a dynamic/static translation process. The average translation overhead of the dynamic binary translation solution is 3.7x higher than the average static approach overhead. The 99%-quantile in the dynamic migration approach is almost 13x higher than that in the static approach and 3.9x higher than the dynamic average execution time. The 99%-quantile in the static approach is just 1.1x higher than the static average execution time. The standard deviation is similar in both migration approaches, 58.9% of the average in the dynamic vs. 51.5% in the static approach.

	Execution time (ns)				
	min	**max**	**avg**	**std**	**99%-quantile**
Dynamic	8342	358739	10043,33	5913,59	38860,76
Static	2558	330480	2751,55	1415,85	2985,18

frequent in the dynamic migration approach. This is reflected on the 99%-quantile, which greatly differs form the average.

To get a better knowledge about how each migration approach performs depending on the characteristics of the translated binary, the following subsection provide a Static vs. Dynamic re-targeting comparative analysis.

7.3.4 Static vs. Dynamic migration

The comparative analysis is performed based on the execution time results obtained from running a WCET representative benchmark suite on top of the legacy and new hardware platforms. The benchmarks are first compiled for the legacy architecture and then translated following static and dynamic migration approaches.

In order to solve scaling problems, results have been clustered into 4 different graphs, see Figure 7.4. These graphs show a comparison between the average value and 99%-quantile (overlapped) of the measured execution time on the new ISAs. Moreover, the standard deviation is represented as an error bar on the average value. Each graph shows the timing results obtained for the dynamic and static migration approaches.

Results show that most of the benchmarks that have been analysed run faster applying the static translation approach (4.6x faster on average), which might be due to the following two reasons: (1) the dynamic approach has heavy run-time overhead, including code translation/optimization and run-time management; and (2) due to the fact that optimisation time counts on execution time, the dynamic approach does not apply aggressive optimization, which leads to worse code quality. However, we did not find any relationship between the benchmark characteristics and the average dynamic/static execution time ratio. As a general rule, the shorter the benchmark execution time, the higher the 99%-quantile/average ratio in the dynamic approach, which goes from 1.03 on long-running to 3.74 on short-running benchmarks. In fact, QEMU's run-time overhead is significant on short-running benchmarks, whereas it is not so, or at least not that much significant, on long-running benchmarks. Moreover, from analysing the average execution time ratio between dynamically translated binaries running on the new hardware and legacy binaries running on the legacy hardware, it can be appreciated that the 10 slowest benchmarks (bs, fac, fdct, fibcall, janne_complex, lcdnum, minver, qsort_exam, select, statemate) are mainly composed of control flow statements and simple computations, except for fdct. Whereas from the analysis of the average execution time ration between statically translated binaries running on the new hardware

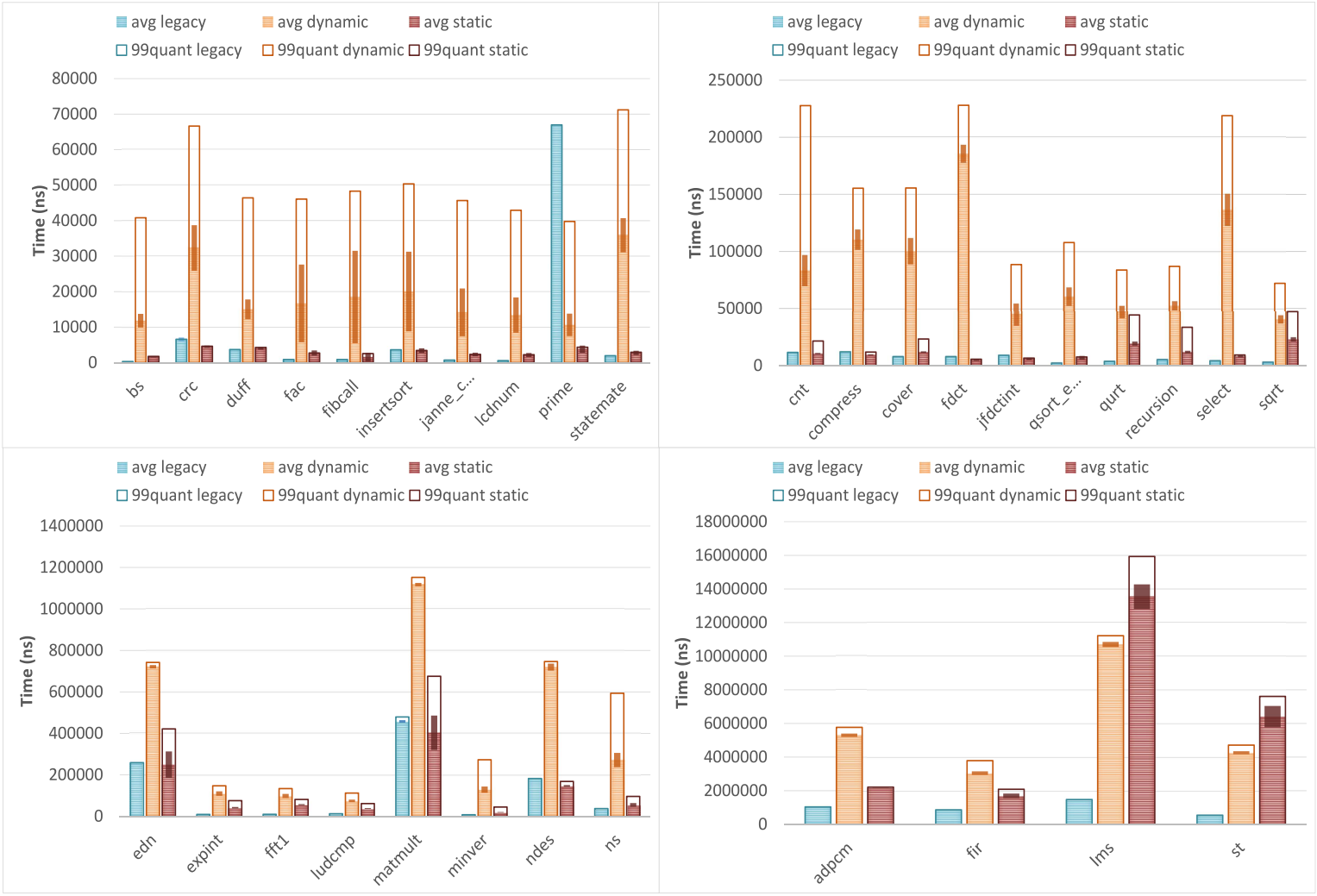


Figure 7.4.: Timing results of benchmarks running on the legacy and new HW platforms: static vs. dynamic translation. Average and 99%-quantile execution time results are shown overlapped and standard deviation is shown as an error bar on average. Most of the benchmarks (except for lms and st) run faster applying the static translation method. The shorter the benchmark execution time, the higher the 99%-quantile/average ratio in the dynamic approach. The slowest (with respect to execution time on the legacy processor) dynamically translated benchmarks are mainly composed of control flow statements (e.g., bs, lcdnum, qsort_exam, select). The slowest (with respect to execution time on the legacy processor) statically translated benchmarks are mainly composed of control flow statements (e.g., bs, lcdnum) and complex floating point computations (e.g., fft1, lms, sqrt, st).

and legacy binaries running on the legacy processor, it can be appreciated that among the 10 slowest benchmarks (bs, expint, fft1, janne_complex, lcdnum, lms, ludcmp, qurt, sqrt, st), some are mainly composed of control flow statements and little computations (e.g., bs, lcdnum), but many others are mainly composed of complex computations (e.g., fft1, lms, sqrt, st). However, these slow benchmarks mainly composed of complex computations, share a common characteristic: they all contains floating point operations. In fact, benchmarks with complex floating point operations (e.g., fft1, lms, ludcmp, sqrt, st) have the lowest average dynamic/static ratio.

7.3.5 Summary

The feasibility study analyses the suitability of QEMU (dynamic) and Rev.ng (static) machine adaptable binary translation tools for their use in a real-time legacy software migration process.

From the experimental result analysis, it can be concluded that among the proposed migration approaches, the static is the most appropriate method to port short-running real-time legacy code. It can provide, compared to the dynamic proposal, lower average execution time and 99%-quantile on every benchmark that has been analysed. Whereas the dynamic approach might be appropriate to port real-time legacy code with long periods (over 0,01 s) and mainly composed of floating point complex computations, since the translation and optimization overhead is not that significant on long-running benchmarks and the static approach implies a great slowdown on benchmarks mainly composed of complex floating point computations.

7.4 Block-Level Timing Enforcement Assessment

The timing enforcement solution presented in Section 5.2.2, which consist of a block-level source code annotation mechanism is being assessed in this section. This evaluation process is accomplished through the example, industrial, and multirotor applications. To this end, the first subsection covers the evaluation set-up, followed by the corresponding result's analysis, which consists of the evaluation of the functional as well as timing behaviour. The timing enforcement assessment is completed with the conclusions reached through the evaluation process.

7.4.1 Evaluation set-up

In order to asses the set of temporal constructs implemented as a block-level timing enforcement mechanism, different applications have been selected (see Section 7.2). For each of the applications the functional as well as the timing behaviour have been assessed. The timing behaviour assessment concerns about the timing behaviour preservation after the timing annotation process, whereas the functional behaviour assessment checks that functional properties are preserved after the block-level timing annotations has been integrated into the application.

The ARM Cortex-A9 has been considered as the legacy processor, so the timing enforcement assessment is carried out on the ZC702 EB (see Section 6.1). As described in Section 6.6, each application is systematically annotated with timing control blocks [7]. The annotated application is linked against the Linux version of the TMCB library and compiled for the ARMv7-A ISA. Then, as described in Section 6.7.1, to test the timing behaviour the annotated code is systematically transformed into formal timing specifications. These timing specifications, together with the time traces that the annotated binary generates when running on top of the ARM Cortex-A9 processor (running Preempt-RT Linux) are feed to the MULTIC tool and thus the timing behaviour is validated. Whereas to test the functional behaviour, as described in Section 6.7.2, the annotated binary running on the ARM Cortex-A9 processor (running Preempt-RT Linux at a 666 MHz operation frequency) is feed with the reference input data and the obtained output data is compared to the reference output data to check the functional behaviour.

To get the reference data, the example application (without annotations) runs on the ARM Cortex-A9 processor and the reference output variable bit toggling sequence is obtained. Then, the control block annotated example application shown in Listing 7.1, which was obtained as a result of the lifting of timing properties, runs on top of the ARM Cortex-A9 processor. The execution lasts for 50000 time steps (statistically representative enough), and the corresponding functional and timing data is collected.

For the industrial application, the application (without annotations) running on top of the ARM Cortex-A9 processor is feed with reference input data that was provided by an expert group and the control variables are observed to collect the reference output data over a 5000 time step execution. Then, the control block annotated industrial application shown in Listing 7.2, which was obtained as a result of the

[7] To annotate the example application every type of timing control block has been used, whereas to annotate the industrial and multirotor applications just the PET blocks is needed.

```
void main() {
  initialization();
  BLOCK_PET(20_ms) {
    BLOCK_FET(10_ms) {
      BLOCK_BET(5_ms) {
        toggle_bit(1, &output_var); //f1
      }
      BLOCK_BET(5_ms) {
        toggle_bit(2, &output_var); //f2
      }
    }
    BLOCK_FET(5_ms) {
      BLOCK_PNET(3_ms, 2, 0) {
        toggle_bit(3, &output_var); //f3
      }
      BLOCK_PNET(3_ms, 2, 1) {
        toggle_bit(4, &output_var); //f4
      }
      BLOCK_BET(2_ms) {
        toggle_bit(5, &output_var); //f5
      }
    }
    BLOCK_BET(2_ms) {
      toggle_bit(6, &output_var); //f6
    }
    BLOCK_BET(3_ms) {
      toggle_bit(7, &output_var); //f7
    }
  }
}
```

Listing 7.1: Example application annotated with time control blocks.

lifting of timing properties, runs on top of the ARM Cortex-A9 processor. As stated before, the execution lasts for 5000 time steps (statistically representative enough), and the corresponding functional and timing data is collected.

```
int main(int argc, char **argv) {
  process_args(argc, argv);
  allocate_memory();
  init_app();
  BLOCK_PET(1_ms) {
    read_inputs();
    execute_app();
    write_outputs();
  }
}
```

Listing 7.2: Industrial application annotated with time control blocks.

For multirotor application, again the application (without annotations) running on top of the ARM Cortex-A9 processor is feed with the reference input data, generated through a flight simulation program i.e. AeroSIM RC [3], and control variables are observed to collect the reference output data. The scenario behind these data is that the multirotor takes off, hoovers for a while and then lands, which covers an execution of about 3000 time steps. Then, the control block annotated multirotor application shown in Listing 7.3, which was obtained as a result of the lifting of

timing properties, runs on top of the ARM Cortex-A9 processor. As stated before, the execution lasts for about 3000 time steps (statistically representative enough), and the corresponding functional and timing data is collected.

```
1 void main(void)
2 {
3   platform_init();
4   BLOCK_PET(2_ms)
5   {
6     platform_execute();
7   }
8 }
```

Listing 7.3: Multirotor application annotated with time control blocks.

7.4.2 Timing test

The timing test validates the timing behaviour of annotated applications (example – see Listing 7.1, industrial – see Listing 7.2, and multirotor – see Listing 7.3) running on the ARM Cortex-A9 processor. Using the MULTIC tool, the time traces that each of the annotated applications generates at run-time are validated against the ideal component-contract structure for each application (example – see Figure 7.5, industrial – see Figure 7.6, and multirotor – see Figure 7.7) obtained as a result of the lifting of timing properties.

Example application – ideal contracts

Table 7.3 shows the time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.5).

Table 7.3.: Time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.5). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET20.Entry 0 ns	Should occur within [0,0] ns.	✓
FET21.Entry 504825 ns	Should occur within [0,0] ns.	✗
BET22.Entry 580910 ns	Should occur within [0,0] ns.	✗
BET22.Exit 706839 ns	Should occur within [580910,5580910] ns.	✓
BET26.Entry 761326 ns	Should occur within [0,5000000] ns.	✓

Continued on next page

Table 7.3 – *Continued from previous page*

Time Trace	Contract limitation	Pass/Fail
BET26.PEntry 504825 ns	Should occur within [0,0] ns.	✗
BET26.Exit 901679 ns	Should occur within [504825,10504825] ns	✓
FET21.Exit 9987845 ns	Should occur within [10504825,10504825] ns.	✗
FET31.Entry 10070300 ns	Should occur within [10000000,10000000] ns.	✗
PNET32.Entry 10131856 ns	Should occur within [10000000,10000000] ns.	✗
PNET32.Exit 10248073 ns	Should occur within [10131856,13131856] ns.	✓
BET40.Entry 10304820 ns	Should occur within [10000000,13000000] ns.	✓
BET40.PEntry 10070300 ns	Should occur within [10000000,10000000] ns.	✗
BET40.Exit 10418145 ns	Should occur within [10304820,15070300] ns.	✓
FET31.Exit 14497737 ns	Should occur within [15070300,15070300] ns.	✗
BET45.Entry 14593920 ns	Should occur within [15000000,15000000] ns.	✗
BET45.Exit 14716674 ns	Should occur within [14593920,16593920] ns.	✓
BET49.Entry 14769626 ns	Should occur within [15000000,17000000] ns.	✗
BET49.PEntry 0 ns	Should occur within [0,0] ns.	✓
BET49.Exit 14907534 ns	Should occur within [15000000,19769626] ns.	✗
PET20.Exit 19989077 ns	Should occur within [20000000,20000000] ns.	✗
PET20.Entry 20052985 ns	Should occur within [20000000,20000000] ns.	✗
FET21.Entry 20112780 ns	Should occur within [20504825,20504825] ns.	✗
BET22.Entry 20171498 ns	Should occur within [20580910,20580910] ns.	✗
BET22.Exit 20284313 ns	Should occur within [20171498,25171498] ns.	✓
BET26.Entry 20337595 ns	Should occur within [15761326,25761326] ns.	✓
BET26.PEntry 20112780 ns	Should occur within [20504825,20504825] ns.	✗
BET26.Exit 20449783 ns	Should occur within [20337595,30112780] ns.	✓
FET21.Exit 29529340 ns	Should occur within [30112780,30112780] ns.	✗
FET31.Entry 29600643 ns	Should occur within [30070300,30070300] ns.	✗
PNET36.Entry 29663642 ns	Should occur within [30000000,30000000] ns.	✗
PNET36.Exit 29779277 ns	Should occur within [29663642,32663642] ns.	✓
BET40.Entry 29832028 ns	Should occur within [27304820,33304820] ns.	✓
BET40.PEntry 29600643 ns	Should occur within [30070300,30070300] ns.	✗
BET40.Exit 29944063 ns	Should occur within [29832028,34600643] ns.	✓
FET31.Exit 34023382 ns	Should occur within [34600643,34600643] ns.	✗
BET45.Entry 34083446 ns	Should occur within [34593920,34593920] ns.	✗
BET45.Exit 34195623 ns	Should occur within [34083446,36083446] ns.	✓
BET49.Entry 34248931 ns	Should occur within [32769626,36769626] ns.	✓
BET49.PEntry 20052985 ns	Should occur within [20000000,20000000] ns.	✗
BET49.Exit 34382318 ns	Should occur within [35052985,39248931] ns.	✗
PET20.Exit 39461774 ns	Should occur within [40052985,40052985] ns.	✗

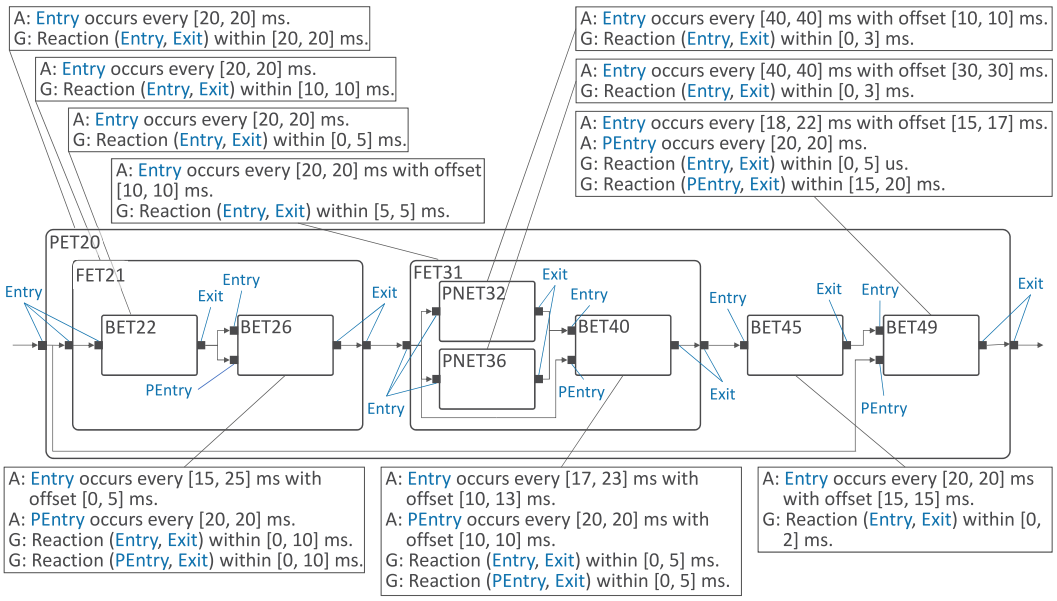


Figure 7.5.: Ideal component-contract structure for the annotated example application. Each component has its corresponding contract. Contracts describe the assumptions and guarantees observable at ports (named in blue colour). Contacts describe an ideal behaviour and therefore do not describe possible time variations (jitter) present in real a scenario.

Industrial application – ideal contracts

Table 7.4 shows the time traces generated by the annotated industrial application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.6).

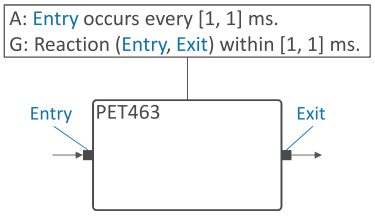


Figure 7.6.: Ideal component-contract structure for the annotated industrial application. The only component (PET463) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract describes an ideal behaviour and therefore does not describe possible time variations (jitter) present in real a scenario.

Table 7.4.: Time traces generated by the annotated industrial application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.6). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET463.Entry 0 ns	Should occur within [0,0] ns.	✓
PET463.Exit 1184268 ns	Should occur within [1000000,1000000] ns.	✗
PET463.Entry 1250930 ns	Should occur within [1000000,1000000] ns.	✗
PET463.Exit 2290602 ns	Should occur within [2250930,2250930] ns.	✗
PET463.Entry 2368367 ns	Should occur within [2250930,2250930] ns.	✗
PET463.Exit 3397243 ns	Should occur within [3368367,3368367] ns	✗
PET463.Entry 3464529 ns	Should occur within [3368367,3368367] ns.	✗
PET463.Exit 4495282 ns	Should occur within [4464529,4464529] ns.	✗
PET463.Entry 4558500 ns	Should occur within [4464529,4464529] ns.	✗
PET463.Exit 5588914 ns	Should occur within [5495282,5495282] ns.	✗
PET463.Entry 5665185 ns	Should occur within [5495282,5495282] ns.	✗
PET463.Exit 6695552 ns	Should occur within [6665185,6665185] ns.	✗
PET463.Entry 6759117 ns	Should occur within [6665185,6665185] ns.	✗
PET463.Exit 7789133 ns	Should occur within [7759117,7759117] ns.	✗
PET463.Entry 7851112 ns	Should occur within [7759117,7759117] ns.	✗
PET463.Exit 8880746 ns	Should occur within [8851112,8851112] ns.	✗
PET463.Entry 8942968 ns	Should occur within [8851112,8851112] ns.	✗
PET463.Exit 9972494 ns	Should occur within [9942968,9942968] ns.	✗
PET463.Entry 10034665 ns	Should occur within [9942968,9942968] ns.	✗
PET463.Exit 11064074 ns	Should occur within [11034665,11034665] ns.	✗

Multirotor application – ideal contracts

Table 7.5 shows the time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.7).

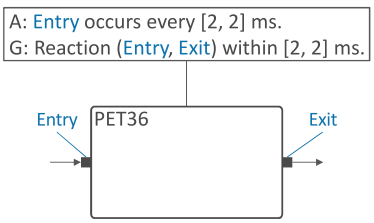


Figure 7.7.: Ideal component-contract structure for the annotated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract describes an ideal behaviour and therefore does not describe possible time variations (jitter) present in real a scenario.

Table 7.5.: Time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 7.7). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET36.Entry 0 ns	Should occur within [0,0] ns.	✓
PET36.Exit 2042507 ns	Should occur within [2000000,2000000] ns.	✗
PET36.Entry 2122859 ns	Should occur within [2000000,2000000] ns.	✗
PET36.Exit 4155069 ns	Should occur within [4122859,4122859] ns.	✗
PET36.Entry 4221302 ns	Should occur within [4122859,4122859] ns.	✗
PET36.Exit 6251056 ns	Should occur within [6221302,6221302] ns	✗
PET36.Entry 6313935 ns	Should occur within [6221302,6221302] ns.	✗
PET36.Exit 8344471 ns	Should occur within [8313935,8313935] ns.	✗
PET36.Entry 8422069 ns	Should occur within [8313935,8313935] ns.	✗
PET36.Exit 10452413 ns	Should occur within [10422069,10422069] ns.	✗
PET36.Entry 10515106 ns	Should occur within [10422069,10422069] ns.	✗
PET36.Exit 12545624 ns	Should occur within [12515106,12515106] ns.	✗
PET36.Entry 12607801 ns	Should occur within [12515106,12515106] ns.	✗
PET36.Exit 14636994 ns	Should occur within [14607801,14607801] ns.	✗
PET36.Entry 14698283 ns	Should occur within [14607801,14607801] ns.	✗
PET36.Exit 16727964 ns	Should occur within [16698283,16698283] ns.	✗
PET36.Entry 16790705 ns	Should occur within [16698283,16698283] ns.	✗
PET36.Exit 18820345 ns	Should occur within [18790705,18790705] ns.	✗
PET36.Entry 18893214 ns	Should occur within [18790705,18790705] ns.	✗
PET36.Exit 20923498 ns	Should occur within [20893214,20893214] ns.	✗

Given that the contracts presented in Figure 7.5, Figure 7.6, and Figure 7.7 are ideal and therefore do not consider any timing variation cause by the overhead of time control block management or by the underlying OS and hardware platform itself, a great percentage of the traces did not pass the corresponding contract (about 60% of the time traces for the example application, and about 99% for the industrial and multirotor applications). In order to cover the timing deviations present in a real scenario, contracts are adjusted as follows:

- Adjust the offset in some of the assumptions (following a repetition pattern) to cover little variations on the time distance from the start of execution to the first occurrence of an event on a particular port.
- Adjust the interval in some of the assumptions (following a repetition pattern) to cover little variations on the time distance between repetitive event occurrences on a particular port.

- Adjust the interval in some of the guarantees (following a causal reaction pattern) to cover little variations on the time distance between causally related events on input/output ports.

Example application – annotation adjusted contracts

Figure 7.8 shows the adjusted component-contract structure for the annotated example application. Changes in the contracts with respect to the ideal component-contract structure in Figure 7.5 are shown in bold. Up to time steps 1600 adjustments had to be done on the contracts, then the example application run for over 5000 time steps without failing any contract. Table 7.6 shows the time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure presented in Figure 7.8. Changes in the time interval with respect to the validation against the ideal component-contract structure (see Table 7.3) are shown in bold.

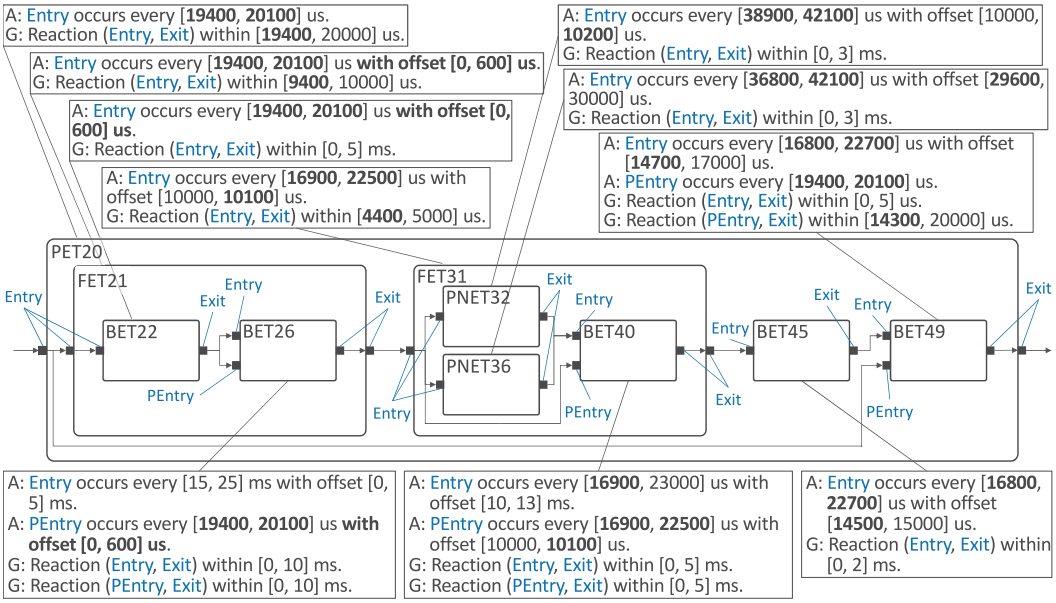


Figure 7.8.: Adjusted component-contract structure for the annotated example application. Each component has its corresponding contract. Contracts describe the assumptions and guarantees observable at input/output ports (named in blue colour). Contracts have been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the annotated example application running on the ARM Cortex-A9 processor. Changes in the contracts with respect to the ideal component-contract structure in Figure 7.5 are shown in bold.

Table 7.6.: Time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure (see Figure 7.8). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with ideal component-contract structure (see Table 7.3) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET20.Entry 0 ns	Should occur within [0,0] ns.	✓
FET21.Entry 504825 ns	Should occur within [0,**600000**] ns.	✓
BET22.Entry 580910 ns	Should occur within [0,**600000**] ns.	✓
BET22.Exit 706839 ns	Should occur within [580910,5580910] ns.	✓
BET26.Entry 761326 ns	Should occur within [0,5000000] ns.	✓
BET26.PEntry 504825 ns	Should occur within [0,**600000**] ns.	✓
BET26.Exit 901679 ns	Should occur within [761326,10504825] ns	✓
FET21.Exit 9987845 ns	Should occur within [**9904825**,10504825] ns.	✓
FET31.Entry 10070300 ns	Should occur within [10000000,**10100000**] ns.	✓
PNET32.Entry 10131856 ns	Should occur within [10000000,**10200000**] ns.	✓
PNET32.Exit 10248073 ns	Should occur within [10131856,13131856] ns.	✓
BET40.Entry 10304820 ns	Should occur within [10000000,13000000] ns.	✓
BET40.PEntry 10070300 ns	Should occur within [10000000,**10100000**] ns.	✓
BET40.Exit 10418145 ns	Should occur within [10304820,15070300] ns.	✓
FET31.Exit 14497737 ns	Should occur within [**14470300**,15070300] ns.	✓
BET45.Entry 14593920 ns	Should occur within [**14500000**,15000000] ns.	✓
BET45.Exit 14716674 ns	Should occur within [14593920,16593920] ns.	✓
BET49.Entry 14769626 ns	Should occur within [**14700000**,17000000] ns.	✓
BET49.PEntry 0 ns	Should occur within [0,0] ns.	✓
BET49.Exit 14907534 ns	Should occur within [**14769626**,19769626] ns.	✓
PET20.Exit 19989077 ns	Should occur within [**19400000**,20000000] ns.	✓
PET20.Entry 20052985 ns	Should occur within [**19400000**,**20100000**] ns.	✓
FET21.Entry 20112780 ns	Should occur within [**19904825**,**20604825**] ns.	✓
BET22.Entry 20171498 ns	Should occur within [**19980910**,**20680910**] ns.	✓
BET22.Exit 20284313 ns	Should occur within [20171498,25171498] ns.	✓
BET26.Entry 20337595 ns	Should occur within [15761326,25761326] ns.	✓
BET26.PEntry 20112780 ns	Should occur within [**19904825**,**20604825**] ns.	✓
BET26.Exit 20449783 ns	Should occur within [20337595,30112780] ns.	✓
FET21.Exit 29529340 ns	Should occur within [**29512780**,30112780] ns.	✓
FET31.Entry 29600643 ns	Should occur within [**26970300**,**32570300**] ns.	✓
PNET36.Entry 29663642 ns	Should occur within [**29600000**,30000000] ns.	✓
PNET36.Exit 29779277 ns	Should occur within [29663642,32663642] ns.	✓
BET40.Entry 29832028 ns	Should occur within [**27204820**,33304820] ns.	✓
BET40.PEntry 29600643 ns	Should occur within [**26970300**,**32570300**] ns.	✓
BET40.Exit 29944063 ns	Should occur within [29832028,34600643] ns.	✓
FET31.Exit 34023382 ns	Should occur within [**34000643**,34600643] ns.	✓
BET45.Entry 34083446 ns	Should occur within [**31393920**,**37293920**] ns.	✓
BET45.Exit 34195623 ns	Should occur within [34083446,36083446] ns.	✓

Continued on next page

Table 7.6 – *Continued from previous page*

Time Trace	Contract limitation	Pass/Fail
BET49.Entry 34248931 ns	Should occur within [**31569626**,**37469626**] ns.	✓
BET49.PEntry 20052985 ns	Should occur within [**19400000**,**20100000**] ns.	✓
BET49.Exit 34382318 ns	Should occur within [**34352985**,39248931] ns.	✓
PET20.Exit 39461774 ns	Should occur within [**39452985**,40052985] ns.	✓

Industrial application – annotation adjusted contracts

Figure 7.9 shows the adjusted component-contract structure for the annotated industrial application. Changes in the contracts with respect to the ideal component-contract structure in Figure 7.6 are shown in bold. On the first two time steps adjustments had to be done on the contract to cover the previously mentioned needs. The interval in the assumption is set to $[1000, 1300]\ us$, whereas the interval in the guarantee is set to $[1000, 1200]\ us$. Then, the industrial application runs for over 5000 time steps without any further failure. Table 7.7) shows the time traces generated by the annotated industrial application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure presented in Figure 7.9. Changes in the time interval with respect to the validation against the ideal component-contract structure (see Table 7.4) are shown in bold.

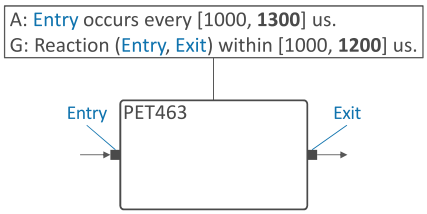


Figure 7.9.: Adjusted component-contract structure for the annotated industrial application. The only component (PET463) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the annotated industrial application running on the ARM Cortex-A9 processor. Changes in the contracts with respect to the ideal component-contract structure in Figure 7.6 are shown in bold.

Table 7.7.: Time traces generated by the annotated industrial application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure (see Figure 7.9). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with ideal component-contract structure (see Table 7.4) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET463.Entry 0 ns	Should occur within [0,0] ns.	✓
PET463.Exit 1184268 ns	Should occur within [1000000,**1200000**] ns.	✓
PET463.Entry 1250930 ns	Should occur within [1000000,**1300000**] ns.	✓
PET463.Exit 2290602 ns	Should occur within [2250930,**2450930**] ns.	✓
PET463.Entry 2368367 ns	Should occur within [2250930,**2550930**] ns.	✓
PET463.Exit 3397243 ns	Should occur within [3368367,**3568367**] ns	✓
PET463.Entry 3464529 ns	Should occur within [3368367,**3668367**] ns.	✓
PET463.Exit 4495282 ns	Should occur within [4464529,**4664529**] ns.	✓
PET463.Entry 4558500 ns	Should occur within [4464529,**4764529**] ns.	✓
PET463.Exit 5588914 ns	Should occur within [5495282,**5695282**] ns.	✓
PET463.Entry 5665185 ns	Should occur within [5495282,**5795282**] ns.	✓
PET463.Exit 6695552 ns	Should occur within [6665185,**6865185**] ns.	✓
PET463.Entry 6759117 ns	Should occur within [6665185,**6965185**] ns.	✓
PET463.Exit 7789133 ns	Should occur within [7759117,**7959117**] ns.	✓
PET463.Entry 7851112 ns	Should occur within [7759117,**8059117**] ns.	✓
PET463.Exit 8880746 ns	Should occur within [8851112,**9051112**] ns.	✓
PET463.Entry 8942968 ns	Should occur within [8851112,**9151112**] ns.	✓
PET463.Exit 9972494 ns	Should occur within [9942968,**10142968**] ns.	✓
PET463.Entry 10034665 ns	Should occur within [9942968,**10242968**] ns.	✓
PET463.Exit 11064074 ns	Should occur within [11034665,**11234665**] ns.	✓

Multirotor application – annotation adjusted contracts

Figure 7.10 shows the adjusted component-contract structure for the annotated multirotor application. Changes in the contracts with respect to the ideal component-contract structure in Figure 7.7 are shown in bold. On the first two time steps adjustments had to be done on the contract to cover the previously mentioned needs. The interval in the assumption is adjusted to $[2000, 2200]\ us$ and the interval y in the guarantee is set to $[2000, 2100]\ us$. Then, the flight control application runs for almost 3000 time steps without any further failure. Table 7.8 shows the time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure presented in Figure 7.10. Changes in the time interval with respect to the validation against the ideal component-contract structure (see Table 7.5) are shown in bold.

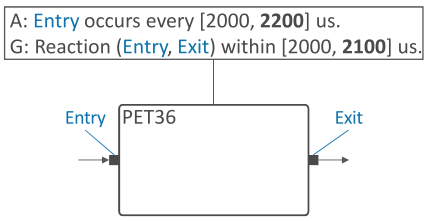


Figure 7.10.: Adjusted component-contract structure for the annotated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the annotated multirotor application running on the ARM Cortex-A9 processor.

Table 7.8.: Time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure (see Figure 7.10). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with ideal component-contract structure (see Table 7.5) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET36.Entry 0 ns	Should occur within [0,0] ns.	✓
PET36.Exit 2042507 ns	Should occur within [2000000,**2100000**] ns.	✓
PET36.Entry 2122859 ns	Should occur within [2000000,**2200000**] ns.	✓
PET36.Exit 4155069 ns	Should occur within [4122859,**4222859**] ns.	✓
PET36.Entry 4221302 ns	Should occur within [4122859,**4322859**] ns.	✓
PET36.Exit 6251056 ns	Should occur within [6221302,**6321302**] ns	✓
PET36.Entry 6313935 ns	Should occur within [6221302,**6421302**] ns.	✓
PET36.Exit 8344471 ns	Should occur within [8313935,**8413935**] ns.	✓
PET36.Entry 8422069 ns	Should occur within [8313935,**8513935**] ns.	✓
PET36.Exit 10452413 ns	Should occur within [10422069,**11422069**] ns.	✓
PET36.Entry 10515106 ns	Should occur within [10422069,**12422069**] ns.	✓
PET36.Exit 12545624 ns	Should occur within [12515106,**13515106**] ns.	✓
PET36.Entry 12607801 ns	Should occur within [12515106,**14515106**] ns.	✓
PET36.Exit 14636994 ns	Should occur within [14607801,**15607801**] ns.	✓
PET36.Entry 14698283 ns	Should occur within [14607801,**16607801**] ns.	✓
PET36.Exit 16727964 ns	Should occur within [16698283,**17698283**] ns.	✓
PET36.Entry 16790705 ns	Should occur within [16698283,**18698283**] ns.	✓
PET36.Exit 18820345 ns	Should occur within [18790705,**19790705**] ns.	✓
PET36.Entry 18893214 ns	Should occur within [18790705,**20790705**] ns.	✓
PET36.Exit 20923498 ns	Should occur within [20893214,**21893214**] ns.	✓

Result analysis

Result show that systematically generated formal timing specifications needed to be adjusted to cover the timing deviations caused by the overhead of time control block management or by the underlying OS and hardware platform itself. After the adjustment of formal timing specifications, time traces generated at runtime by the

annotate example, industrial and multirotor applications fit into the defined timing contracts. Future work considers characterizing the overhead generated by control block management and adjusting the blocks accordingly to, at some point, overcome it (see Section 8.2).

7.4.3 Functional test

The functional test compares the reference output value for control variables with the output value (for those control variables) when running the annotated applications (example – see Listing 7.1, industrial – see Listing 7.2, and multirotor – see Listing 7.3) on the ARM Cortex-A9 processor (using reference input data for state variables).

Example application

Figure 7.11 shows the bit toggling sequence of the output variable on both test scenarios. However, due to scaling problems, the figure depicts just the output variable value for the first 100 time steps.

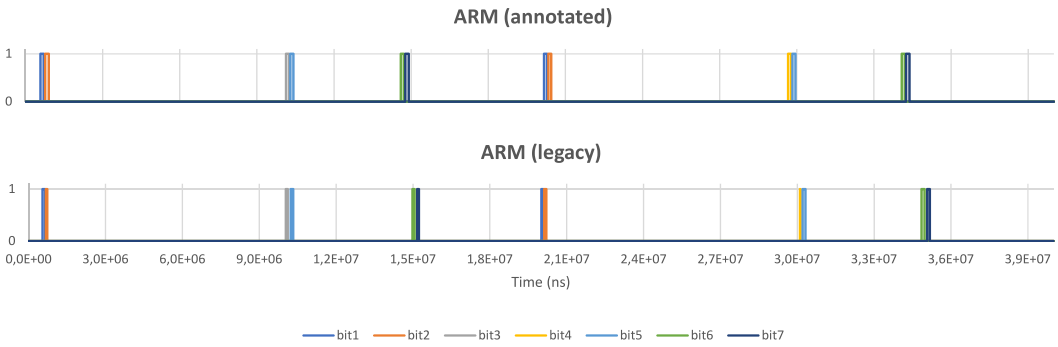
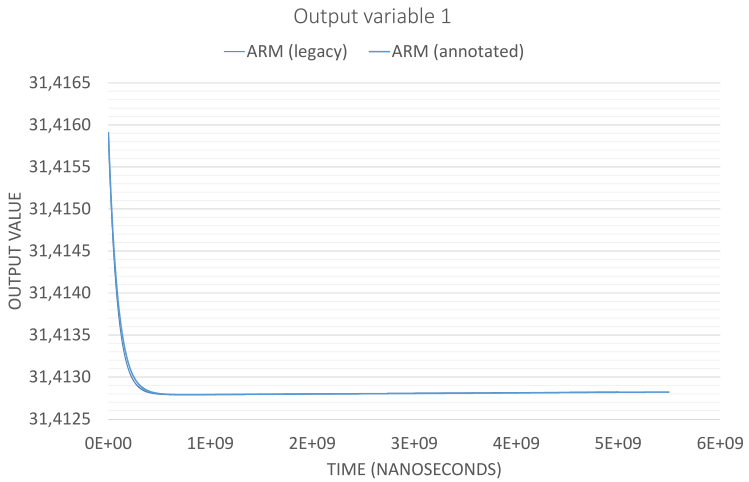


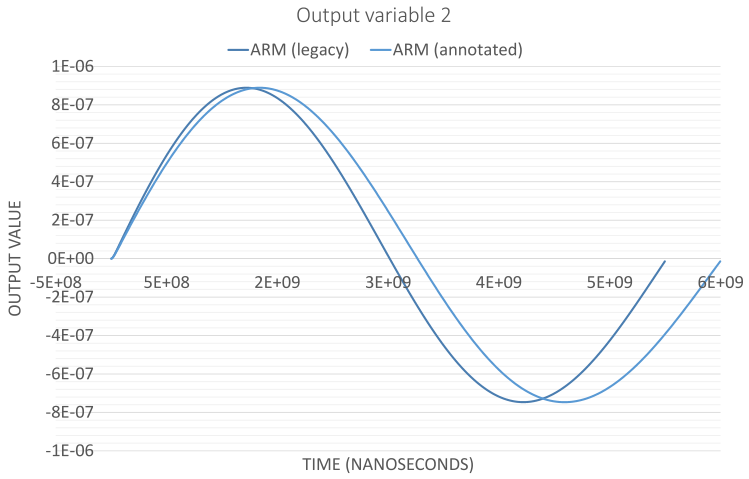
Figure 7.11.: Functional test results for the annotated example application running on the ARM Cortex-A9 processor. For each time step (X-axis) the corresponding output variable value (Y-axis) is shown, both, for the legacy example application (without annotations) as well as for the annotated example application. On both test scenarios (legacy and annotated), the bit toggling sequence is equal, although there is a delay on the annotated application with respect to the legacy application due to time control block management.

Industrial application

Figure 7.12 shows the output values for each of the industrial application control variables on both test scenarios. Control variables are observed for a 5000 time step interval.



(a) Output variable 1



(b) Output variable 2

Figure 7.12.: Functional test results for the annotated industrial application running on the ARM Cortex-A9 processor. (a) shows for each time step (in the X-axis) the corresponding output value of control variable 1 (in the Y-axis), whereas (b) shows for each time step (in the X-axis) the corresponding output value of control variable 2 (in the Y-axis). Both, (a) and (b) show the results obtained for the legacy industrial application (without annotations) as well as for the annotated industrial application. On both test scenario (legacy and annotated), the output value of the control variables is equal, although there is a delay on the annotated application with respect to the legacy application due to time control block management.

Multirotor application

Figure 7.13 shows the output values for each of the multirotor application control variables on both test scenarios. Control variables are observed for an interval of about 3000 time steps (about 6 s simulation).

Result analysis

Results show that the signal observed on every control variable is equivalent (in value) before and after the lifting of timing properties. However, due to time control block management overhead, a delay (with respect to non-annotated code) is observable on every control variable output signal. This delay in the response of the controller might slightly disturb the functional behaviour of the overall control system, therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the lifting of timing properties. Future work considers providing support to port legacy platform I/O port dependent code, which will provide means to perform a functional test on a more realistic scenario (see Section 8.2).

7.4.4 Summary

The block-level timing enforcement assessment, evaluates the lifting of timing properties presented in Section 5.2 through an example, an industrial and a multirotor use-case. The former, is used to evaluate corner-cases, however it is self generated code, therefore, the industrial and multirotor use-cases are used to evaluate the effort of porting real third party legacy code.

According to the timing test, results show that systematically generated formal timing specifications needed to be adjusted to cover timing deviations caused by the overhead of time control block management as well as timing deviations caused by the underlying OS and hardware platform itself. For each case it might be necessary to evaluate if the introduced overhead is affordable. Once timing contracts are adjusted, as shown in Figures 7.8 to 7.10, time traces generated at runtime through the annotated time control blocks (see Tables 7.6 to 7.8) fit into the defined time contacts. Therefore, it can be concluded that the proposed block-level timing enforcement mechanism incurs some overhead, but in turn it can be applied through the lifting process to transform legacy timing properties (implicit on the legacy

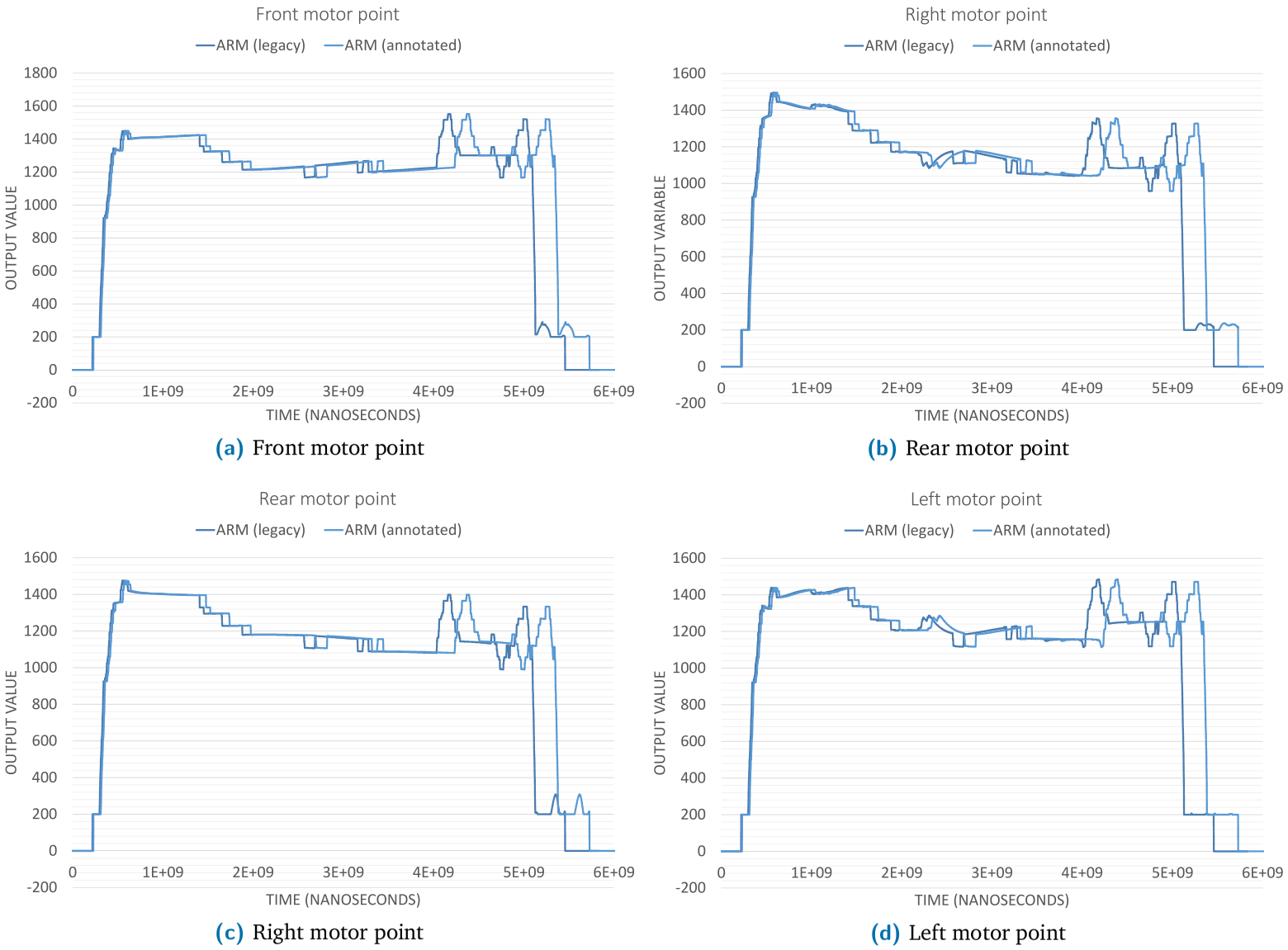


Figure 7.13.: Functional test results for the annotated multirotor application running on the ARM Cortex-A9 processor. (a) shows for each time step (in the X-axis) the corresponding value of motor front point variable (in the Y-axis), (b) shows for each time step (in the X-axis) the corresponding value of motor rear point variable (in the Y-axis), (c) shows for each time step (in the X-axis) the corresponding value of motor right point variable (in the Y-axis), and (d) shows for each time step (in the X-axis) the corresponding value of motor left point variable (in the Y-axis). Every graph, (a), (b), (c), and (d), show the results obtained for the legacy multirotor application (without annotations) as well as for the annotated multirotor application. On both test scenarios (legacy and annotated), the output value of control variables is equal, although there is a delay on the annotated application with respect to the legacy application due to time control block management.

code) into explicit timing properties that will then be preserved on the translation process.

Functional test results (see Figures 7.11 to 7.13) show that the transformations applied on the lifting of timing properties do not disturb the value observed on any of the control variable output signals. However, due to time control block management overhead, a delay is observable in the controller response time with respect to the (original) legacy system, which might slightly disturb the functional behaviour of the overall control system. Therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the lifting of timing properties. As described in Section 8.2, future work considers I/O virtualization in order to provides means for such analysis.

7.5 Timing-aware Static Legacy Software Translation Assessment

This section evaluates the timing equivalent static BT approach described in Section 5.4.1. As the timing enforcement assessment, the assessment of the timing-aware static legacy software translation is accomplished through the example, industrial, and multirotor applications. Therefore, the first subsection covers the evaluation set-up, which is followed by the results analysis section where the functional and timing behaviour is evaluated. To end up, the last section presents the conclusion reached through the evaluation process.

7.5.1 Evaluation set-up

As already state, in order to offer a wide analysis, multiple applications have been selected to assess from a timing and functional perspective the proposed timing-aware static translation approach (see Section 7.2). The timing behaviour assessment concerns about the preservation of the timing behaviour after the timing-aware static translation is accomplished, whereas the functional behaviour assessment checks that the functional behaviour is preserved after the timing-aware static translation process.

As for feasibility study, the ARM Cortex-A9 and the Intel Atom E3866 processors have been selected as source and target respectively. So, the timing-aware translation

assessment is carried out on the ZC702 and Minnowboard EBs (see Section 6.1). As described in Section 6.6, each application is systematically annotated with timing control blocks [7]. The annotated application is linked against the Linux implementation of the TMCB library and compiled for the ARMv7-A ISA. Then, using Rev.ng tool, the annotated ARM binary is statically translated into an equivalent binary for x86 ISA that is executed on top of the Intel Atom E3866 processor (running Preempt-RT Linux). To test the timing behaviour, as described in Section 6.7.1, from the annotated legacy code formal timing specifications are systematically obtained. These timing specifications, which have been adjusted as part of the block-level timing enforcement assessment, are latter used on the MULTIC tool, together with the time traces that the translated binary generates when running on the Intel Atom processor, to validate the timing behaviour. To test the functional behaviour, as described in Section 6.7.2, the translated binary running on the Intel Atom processor is feed with the reference input data and the obtained output data is compared to the reference output data, Section 7.4.1 describes how reference input/output data is obtained for each of the applications (example, industrial and multirotor).

7.5.2 Timing test

The timing test validates the timing behaviour of translated applications (example – see Listing 7.1, industrial – see Listing 7.2, and multirotor – see Listing 7.3) running on the Intel Atom E3866 processor. Using the MULTIC tool, the time traces that each of the translated applications generates at run-time are validated against the component-contract structure for each application (example – see Figure 7.8, industrial – see Figure 7.9, and multirotor – see Figure 7.10), obtained as a result of the lifting of timing properties and adjusted as part of the timing enforcement assessment.

Example application – annotation adjusted contracts

Table 7.9 shows the time traces generated by the translated example application running on the Intel Atom E3866 processor validated against the component-contract structure presented in Figure 7.8, adjusted as part of the timing enforcement assessment.

Table 7.9.: Time traces generated by the translated example application running on the Intel Atom E3866 processor validated against the (before translation) adjusted component-contract structure (see Figure 7.8). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET20.Entry 0 ns	Should occur within [0,0] ns.	✓
FET21.Entry 860897 ns	Should occur within [0,600000] ns.	✗
BET22.Entry 1093058 ns	Should occur within [0,600000] ns.	✗
BET22.Exit 1447619 ns	Should occur within [1093058,6093058] ns.	✓
BET26.Entry 1597033 ns	Should occur within [0,5000000] ns.	✓
BET26.PEntry 860897 ns	Should occur within [0,600000] ns.	✗
BET26.Exit 1966338 ns	Should occur within [1597033,10860897] ns	✓
FET21.Exit 11138808 ns	Should occur within [10260897,10860897] ns.	✗
FET31.Entry 11445947 ns	Should occur within [10000000,10100000] ns.	✗
PNET32.Entry 11644891 ns	Should occur within [10000000,10200000] ns.	✗
PNET32.Exit 12063298 ns	Should occur within [11644891,14644891] ns.	✓
BET40.Entry 12233075 ns	Should occur within [10000000,13000000] ns.	✓
BET40.PEntry 11445947 ns	Should occur within [10000000,10100000] ns.	✗
BET40.Exit 12580840 ns	Should occur within [12233075,16445947] ns.	✓
FET31.Exit 17023455 ns	Should occur within [15845947,16445947] ns.	✗
BET45.Entry 17404573 ns	Should occur within [14500000,15000000] ns.	✗
BET45.Exit 17816493 ns	Should occur within [17404573,19404573] ns.	✓
BET49.Entry 17987424 ns	Should occur within [14700000,17000000] ns.	✗
BET49.PEntry 0 ns	Should occur within [0,0] ns.	✓
BET49.Exit 18307373 ns	Should occur within [17987424,20000000] ns.	✓
PET20.Exit 20486905 ns	Should occur within [19400000,20000000] ns.	✗
PET20.Entry 20932500 ns	Should occur within [19400000,20100000] ns.	✗
FET21.Entry 21241476 ns	Should occur within [20260897,20960897] ns.	✗
BET22.Entry 21532219 ns	Should occur within [20493058,21193058] ns.	✗
BET22.Exit 22164893 ns	Should occur within [21532219,26532219] ns.	✓
BET26.Entry 22428261 ns	Should occur within [16597033,26597033] ns.	✓
BET26.PEntry 21241476 ns	Should occur within [20260897,20960897] ns.	✓
BET26.Exit 23046048 ns	Should occur within [22428261,31241476] ns.	✓
FET21.Exit 31745743 ns	Should occur within [30641476,31241476] ns.	✗
FET31.Entry 32165890 ns	Should occur within [28345947,33945947] ns.	✓
PNET36.Entry 32493916 ns	Should occur within [29600000,30000000] ns.	✗
PNET36.Exit 33168537 ns	Should occur within [32493916,35493916] ns.	✓
BET40.Entry 33432596 ns	Should occur within [29133075,35233075] ns.	✓
BET40.PEntry 32165890 ns	Should occur within [28345947,33945947] ns.	✓
BET40.Exit 34055520 ns	Should occur within [33432596,37165890] ns.	✓
FET31.Exit 37729554 ns	Should occur within [36565890,37165890] ns.	✗
BET45.Entry 38097877 ns	Should occur within [34204573,40104573] ns.	✓
BET45.Exit 38600576 ns	Should occur within [38097877,40097877] ns.	✓
BET49.Entry 38878960 ns	Should occur within [34787424,40687424] ns.	✓

Continued on next page

Table 7.9 – *Continued from previous page*

Time Trace	Contract limitation	Pass/Fail
BET49.PEntry 20932500 ns	Should occur within [19400000,20100000] ns.	✗
BET49.Exit 39354809 ns	Should occur within [38878960,40932500] ns.	✓
PET20.Exit 41428915 ns	Should occur within [40332500,41032500] ns.	✗

Industrial application – annotation adjusted contracts

Table 7.10 shows the time traces generated by the translated industrial application running on the Intel Atom E3866 processor validated against the component-contract structure presented in Figure 7.9, adjusted as part of the timing enforcement assessment.

Table 7.10.: Time traces generated by the translated industrial application running on the Intel Atom E3866 processor validated against the (before translation) adjusted component-contract structure (see Figure 7.9). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET463.Entry 0 ns	Should occur within [0,0] ns.	✓
PET463.Exit 1197464 ns	Should occur within [1000000,1200000] ns.	✓
PET463.Entry 1402604 ns	Should occur within [1000000,1300000] ns.	✗
PET463.Exit 2476902 ns	Should occur within [2402604,2602604] ns.	✓
PET463.Entry 2782782 ns	Should occur within [2402604,2702604] ns.	✗
PET463.Exit 3838616 ns	Should occur within [3782782,3982782] ns	✓
PET463.Entry 4073170 ns	Should occur within [3782782,4082782] ns.	✓
PET463.Exit 5125884 ns	Should occur within [5073170,5273170] ns.	✓
PET463.Entry 5354859 ns	Should occur within [5073170,5373170] ns.	✓
PET463.Exit 6585172 ns	Should occur within [6354859,6554859] ns.	✗
PET463.Entry 6836977 ns	Should occur within [6354859,6654859] ns.	✗
PET463.Exit 8048998 ns	Should occur within [7836977,8036977] ns.	✗
PET463.Entry 8275385 ns	Should occur within [7836977,8136977] ns.	✗
PET463.Exit 9508556 ns	Should occur within [9275385,9475385] ns.	✗
PET463.Entry 9766759 ns	Should occur within [9275385,9575385] ns.	✗
PET463.Exit 10821137 ns	Should occur within [10766759,10966759] ns.	✓
PET463.Entry 11052332 ns	Should occur within [10766759,11066759] ns.	✓
PET463.Exit 12103321 ns	Should occur within [12052332,12252332] ns.	✓
PET463.Entry 12315870 ns	Should occur within [12052332,12352332] ns.	✓
PET463.Exit 13496474 ns	Should occur within [13315870,13515870] ns.	✓

Multirotor application – annotation adjusted contracts

Table 7.11 shows the time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the component-contract structure presented in Figure 7.10, adjusted as part of the timing enforcement assessment.

Table 7.11.: Time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the (before translation) adjusted component-contract structure (see Figure 7.10). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET36.Entry 0 ns	Should occur within [0,0] ns.	✓
PET36.Exit 2124215 ns	Should occur within [2000000,2100000] ns.	✗
PET36.Entry 2541230 ns	Should occur within [2000000,2200000] ns.	✗
PET36.Exit 4868417 ns	Should occur within [4541230,4641230] ns.	✗
PET36.Entry 5146059 ns	Should occur within [4541230,4741230] ns.	✗
PET36.Exit 7643136 ns	Should occur within [7146059,7246059] ns	✗
PET36.Entry 7935216 ns	Should occur within [7146059,7346059] ns.	✗
PET36.Exit 10450728 ns	Should occur within [9935216,10035216] ns.	✗
PET36.Entry 10743715 ns	Should occur within [9935216,10135216] ns.	✗
PET36.Exit 13303320 ns	Should occur within [12743715,12843715] ns.	✗
PET36.Entry 13596742 ns	Should occur within [12743715,12943715] ns.	✗
PET36.Exit 16099024 ns	Should occur within [15596742,15696742] ns.	✗
PET36.Entry 16501496 ns	Should occur within [15596742,15796742] ns.	✗
PET36.Exit 19017908 ns	Should occur within [18501496,18601496] ns.	✗
PET36.Entry 19434240 ns	Should occur within [18501496,18701496] ns.	✗
PET36.Exit 21984387 ns	Should occur within [21434240,21534240] ns.	✗
PET36.Entry 22414182 ns	Should occur within [21434240,21634240] ns.	✗
PET36.Exit 24975024 ns	Should occur within [24414182,24514182] ns.	✗
PET36.Entry 25401653 ns	Should occur within [24414182,24614182] ns.	✗
PET36.Exit 27949025 ns	Should occur within [27401653,27501653] ns.	✗

The contracts presented in Figure 7.8, Figure 7.9, and Figure 7.10 had already been adjusted, however, they do not consider any timing variation cause by the overhead of the static translation process. Moreover, the code is now running on the new hardware platform, so time variations caused by the underlying OS and hardware platform itself, might vary. This caused many of the traces to fail the corresponding contract (almost 50% of the traces for the example application, about 40% of the traces for the industrial application, and every trace for the multirotor application).

In order to cover the timing deviations caused by the static translation and the new hardware platform, contracts are adjusted as it was done before translation.

Example application – translation adjusted contracts

Figure 7.14 shows the adjusted component-contract structure for the translated example application. Changes in the contracts with respect to the (before translation) adjusted component-contract structure in Figure 7.8 are shown in bold. Up to time step 1400 adjustments had to be done in the contracts, then the example application runs for over 5000 time steps without failing any contract. Table 7.12 shows the time traces generated by the annotated example application running on the Intel Atom E3866 processor validated against the adjusted component-contract structure presented in Figure 7.14. Changes in the time interval with respect to the validation against the (before translation) adjusted component-contract structure (see Table 7.9) are shown in bold.

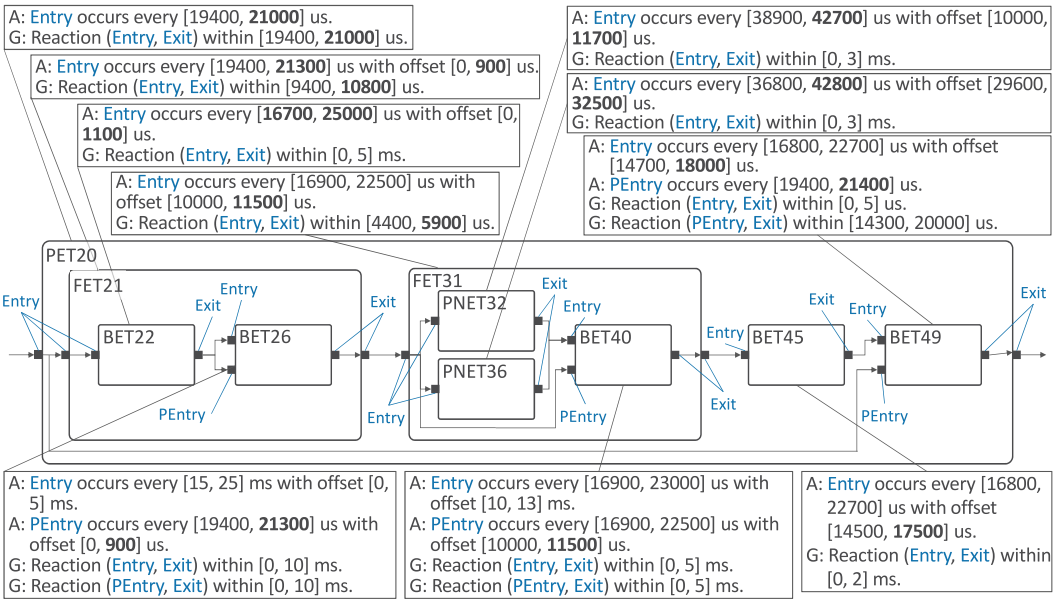


Figure 7.14.: Adjusted component-contract structure for the translated example application. Each component has its corresponding contract. Contracts describe the assumptions and guarantees observable at input/output ports (named in blue colour). Contracts have been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the translated example application running on the Intel Atom E3866 processor. Changes in the contracts with respect to the (before translation) adjusted component-contract structure in Figure 7.8 are shown in bold.

Table 7.12.: Time traces generated by the translated example application running on the Intel Atom E3866 processor validated against the (after translation) adjusted component-contract structure (see Figure 7.14). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with (before translation) adjusted component-contract structure (see Table 7.11) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET20.Entry 0 ns	Should occur within [0,0] ns.	✓
FET21.Entry 860897 ns	Should occur within [0,**900000**] ns.	✓
BET22.Entry 1093058 ns	Should occur within [0,**1100000**] ns.	✓
BET22.Exit 1447619 ns	Should occur within [1093058,6093058] ns.	✓
BET26.Entry 1597033 ns	Should occur within [0,5000000] ns.	✓
BET26.PEntry 860897 ns	Should occur within [0,**900000**] ns.	✓
BET26.Exit 1966338 ns	Should occur within [1597033,10860897] ns	✓
FET21.Exit 11138808 ns	Should occur within [10260897,**11660897**] ns.	✓
FET31.Entry 11445947 ns	Should occur within [10000000,**11500000**] ns.	✓
PNET32.Entry 11644891 ns	Should occur within [10000000,**11700000**] ns.	✓
PNET32.Exit 12063298 ns	Should occur within [11644891,14644891] ns.	✓
BET40.Entry 12233075 ns	Should occur within [10000000,13000000] ns.	✓
BET40.PEntry 11445947 ns	Should occur within [10000000,**11500000**] ns.	✓
BET40.Exit 12580840 ns	Should occur within [12233075,16445947] ns.	✓
FET31.Exit 17023455 ns	Should occur within [15845947,**17345947**] ns.	✓
BET45.Entry 17404573 ns	Should occur within [14500000,**17500000**] ns.	✓
BET45.Exit 17816493 ns	Should occur within [17404573,19404573] ns.	✓
BET49.Entry 17987424 ns	Should occur within [14700000,**18000000**] ns.	✓
BET49.PEntry 0 ns	Should occur within [0,0] ns.	✓
BET49.Exit 18307373 ns	Should occur within [17987424,20000000] ns.	✓
PET20.Exit 20486905 ns	Should occur within [19400000,**21000000**] ns.	✓
PET20.Entry 20932500 ns	Should occur within [19400000,**21000000**] ns.	✓
FET21.Entry 21241476 ns	Should occur within [20260897,**22160897**] ns.	✓
BET22.Entry 21532219 ns	Should occur within [**17793058**,**26093058**] ns.	✓
BET22.Exit 22164893 ns	Should occur within [21532219,26532219] ns.	✓
BET26.Entry 22428261 ns	Should occur within [16597033,26597033] ns.	✓
BET26.PEntry 21241476 ns	Should occur within [20260897,**22160897**] ns.	✓
BET26.Exit 23046048 ns	Should occur within [22428261,31241476] ns.	✓
FET21.Exit 31745743 ns	Should occur within [30641476,**32041476**] ns.	✓
FET31.Entry 32165890 ns	Should occur within [28345947,33945947] ns.	✓
PNET36.Entry 32493916 ns	Should occur within [29600000,**32500000**] ns.	✓
PNET36.Exit 33168537 ns	Should occur within [32493916,35493916] ns.	✓
BET40.Entry 33432596 ns	Should occur within [29133075,35233075] ns.	✓
BET40.PEntry 32165890 ns	Should occur within [28345947,33945947] ns.	✓
BET40.Exit 34055520 ns	Should occur within [33432596,37165890] ns.	✓
FET31.Exit 37729554 ns	Should occur within [36565890,**38065890**] ns.	✓
BET45.Entry 38097877 ns	Should occur within [34204573,40104573] ns.	✓

Continued on next page

Table 7.12 – *Continued from previous page*

Time Trace	Contract limitation	Pass/Fail
BET45.Exit 38600576 ns	Should occur within [38097877,40097877] ns.	✓
BET49.Entry 38878960 ns	Should occur within [34787424,40687424] ns.	✓
BET49.PEntry 20932500 ns	Should occur within [19400000,**21400000**] ns.	✓
BET49.Exit 39354809 ns	Should occur within [38878960,40932500] ns.	✓
PET20.Exit 41428915 ns	Should occur within [40332500,**41932500**] ns.	✓

Industrial application – translation adjusted contracts

Figure 7.15 shows the adjusted component-contract structure for the translated industrial application. Up to time step 2400 adjustments had to be done in the contract to cover the previously mentioned needs. The interval in the assumption is set to $[1, 2]$ ms, whereas the interval in the guarantee is set to $[1000, 1500]$ us. Then, the industrial application runs for over 5000 time steps without any further failure. Table 7.13 shows the time traces generated by the translated industrial application running on the Intel Atom E3866 processor validated against the adjusted component-contract structure presented in Figure 7.15. Changes in the time interval with respect to the validation against the (before translation) adjusted component-contract structure (see Table 7.10) are shown in bold.

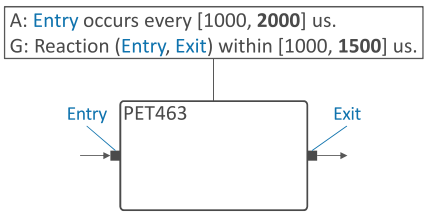


Figure 7.15.: Adjusted component-contract structure for the translated industrial application. The only component (PET463) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the translated industrial application running on the Intel Atom E3866 processor.

Table 7.13.: Time traces generated by the translated industrial application running on the Intel Atom E3866 processor validated against the (after translation) adjusted component-contract structure (see Figure 7.15). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with the (before translation) adjusted component-contract structure (see Table 7.10) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET463.Entry 0 ns	Should occur within [0,0] ns.	✓
PET463.Exit 1197464 ns	Should occur within [1000000,**1500000**] ns.	✓
PET463.Entry 1402604 ns	Should occur within [1000000,**2000000**] ns.	✓
PET463.Exit 2476902 ns	Should occur within [2402604,**2902604**] ns.	✓
PET463.Entry 2782782 ns	Should occur within [2402604,**3402604**] ns.	✓
PET463.Exit 3838616 ns	Should occur within [3782782,**4282782**] ns	✓
PET463.Entry 4073170 ns	Should occur within [3782782,**4782782**] ns.	✓
PET463.Exit 5125884 ns	Should occur within [5073170,**5573170**] ns.	✓
PET463.Entry 5354859 ns	Should occur within [5073170,**6073170**] ns.	✓
PET463.Exit 6585172 ns	Should occur within [6354859,**6854859**] ns.	✓
PET463.Entry 6836977 ns	Should occur within [6354859,**7354859**] ns.	✓
PET463.Exit 8048998 ns	Should occur within [7836977,**8336977**] ns.	✓
PET463.Entry 8275385 ns	Should occur within [7836977,**8836977**] ns.	✓
PET463.Exit 9508556 ns	Should occur within [9275385,**9775385**] ns.	✓
PET463.Entry 9766759 ns	Should occur within [9275385,**10275385**] ns.	✓
PET463.Exit 10821137 ns	Should occur within [10766759,**11266759**] ns.	✓
PET463.Entry 11052332 ns	Should occur within [10766759,**11766759**] ns.	✓
PET463.Exit 12103321 ns	Should occur within [12052332,**12552332**] ns.	✓
PET463.Entry 12315870 ns	Should occur within [12052332,**13052332**] ns.	✓
PET463.Exit 13496474 ns	Should occur within [13315870,**13815870**] ns.	✓

Multirotor application – translation adjusted contracts

Figure 7.16 shows the adjusted component-contract structure for the translated multirotor application. Up to time step 1000 adjustments had to be done in the contract to cover the previously mentioned needs. The interval in the assumption is set to $[2000, 3600]\ ms$, whereas the interval in the guarantee is set to $[2000, 3000]\ us$. Then, the multirotor application runs for over 3000 time steps without any further failure. Table 7.14 shows the time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the adjusted component-contract structure presented in Figure 7.16. Changes in the time interval with respect to the validation against the (before translation) adjusted component-contract structure (see Table 7.11) are shown in bold.

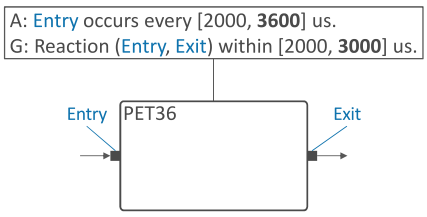


Figure 7.16.: Adjusted component-contract structure for the translated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the translated multirotor application running on the Intel Atom E3866 processor.

Table 7.14.: Time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the (after translation) adjusted component-contract structure (see Figure 7.16). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with the (before translation) component-contract structure (see Table 7.11) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass/Fail
PET36.Entry 0 ns	Should occur within [0,0] ns.	✓
PET36.Exit 2124215 ns	Should occur within [2000000,**3000000**] ns.	✓
PET36.Entry 2541230 ns	Should occur within [2000000,**3600000**] ns.	✓
PET36.Exit 4868417 ns	Should occur within [4541230,**5541230**] ns.	✓
PET36.Entry 5146059 ns	Should occur within [4541230,**6141230**] ns.	✓
PET36.Exit 7643136 ns	Should occur within [7146059,**8146059**] ns	✓
PET36.Entry 7935216 ns	Should occur within [7146059,**8746059**] ns.	✓
PET36.Exit 10450728 ns	Should occur within [9935216,**10935216**] ns.	✓
PET36.Entry 10743715 ns	Should occur within [9935216,**11535216**] ns.	✓
PET36.Exit 13303320 ns	Should occur within [12743715,**13743715**] ns.	✓
PET36.Entry 13596742 ns	Should occur within [12743715,**14343715**] ns.	✓
PET36.Exit 16099024 ns	Should occur within [15596742,**16596742**] ns.	✓
PET36.Entry 16501496 ns	Should occur within [15596742,**17196742**] ns.	✓
PET36.Exit 19017908 ns	Should occur within [18501496,**19501496**] ns.	✓
PET36.Entry 19434240 ns	Should occur within [18501496,**20101496**] ns.	✓
PET36.Exit 21984387 ns	Should occur within [21434240,**22434240**] ns.	✓
PET36.Entry 22414182 ns	Should occur within [21434240,**23034240**] ns.	✓
PET36.Exit 24975024 ns	Should occur within [24414182,**25414182**] ns.	✓
PET36.Entry 25401653 ns	Should occur within [24414182,**26014182**] ns.	✓
PET36.Exit 27949025 ns	Should occur within [27401653,**28401653**] ns.	✓

Result analysis

Result show that formal timing specifications adjusted as part of the timing enforcement assessment needed to be re-adjusted to cover the timing deviations generated due to translation overhead. After the re-adjustment of formal timing specifications, time traces generated at runtime by the translated example, industrial and multirotor

applications fit into the defined timing contracts. However, the static translation process involves a considerable overhead (analysed through the feasibility study – see Section 7.3) which might not be affordable depending on the application to be ported. For each particular case, an expert should assess, considering the incurred overhead, whether the RT legacy software migration solution is acceptable. Future work considers optimizing time control blocks to, at some point, overcome the block management and translation overhead (see Section 8.2).

7.5.3 Functional test

The functional test compares the reference output values for control variables with the output value (for those control variables) when running the systematically annotated and then translated applications (example – see Listing 7.1, industrial – see Listing 7.2, and multirotor – see Listing 7.3) on top of the Intel Atom processor.

Example application

Figure 7.17 shows the bit toggling sequence of the output variable on both test scenarios. However, due to scaling problems, the figure depicts just the output variable value for the first 100 time steps.

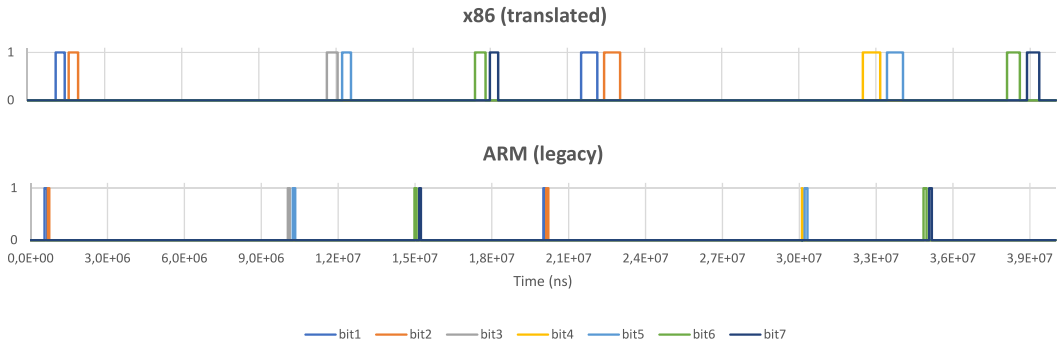


Figure 7.17.: Functional test results for the translated example application running on the Intel Atom E3866 processor. For each time step (X-axis) the corresponding output variable value (Y-axis) is shown, both, for the legacy example application (without annotations) as well as for the translated example application. On both test scenarios (legacy and translated), the bit toggling sequence is equal, although there is a delay on the translated application with respect to the legacy application due to time control block management and static translation overhead.

Industrial application

Figure 7.18 shows the output values for each of the industrial application control variables on both test scenarios. Control variables are observed for a 5000 time step interval.

Multirotor application

Figure 7.19 shows the output values for each of the multirotor control variables on both test scenarios. Control variables are observed for a 3000 time step interval (about 6 s simulation).

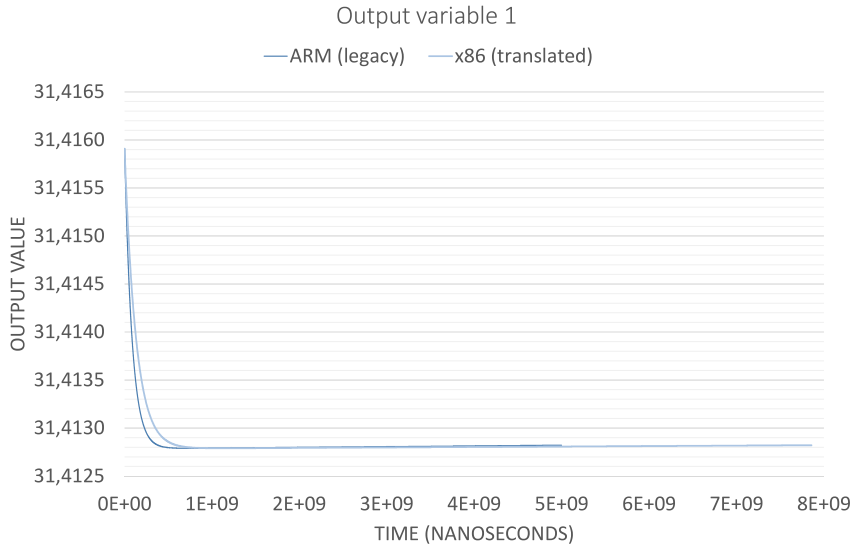
Result analysis

Results show that the signal observed on every control variable is equivalent (in value) before and after the timing-aware migration process. However, due to time control block management and static translation overhead, a delay (with respect to the legacy platform) is observable on every control variable output signal. This delay in the response of the controller might disturb the functional behaviour of the control system, therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the RT legacy software migration process. Future work considers providing support to port legacy platform I/O port dependent code, which will provide means to perform a functional test on a more realistic scenario (see Section 8.2).

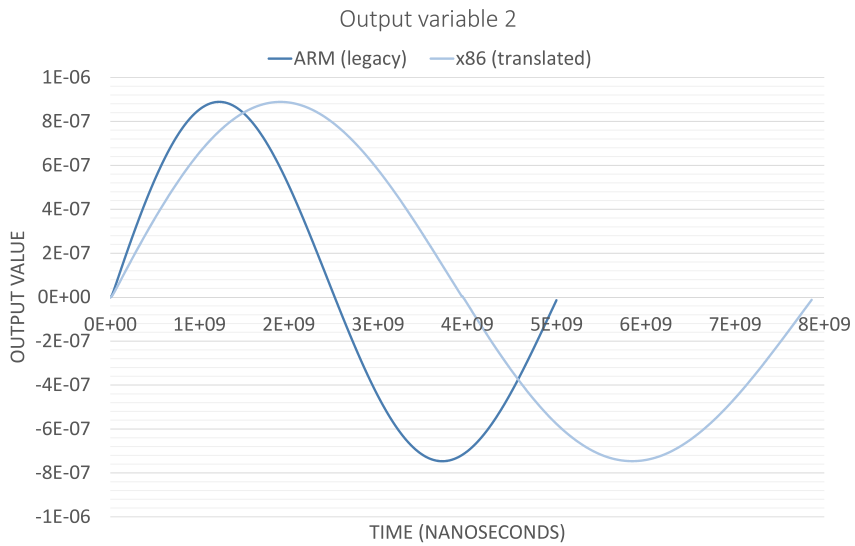
7.5.4 Summary

The timing-aware static legacy software translation assessment, evaluates the timing block handling within the static BT process (presented in Section 5.4.1) through an example, an industrial and a multirotor use-case. The former, is used to evaluate corner-cases, however it is self generated code, therefore, the industrial and multirotor use-cases are used to evaluate the effort of porting real third party legacy code.

According to the timing test, results show that systematically generated formal timing specifications needed to be adjusted to cover timing deviations caused by the overhead of the static translation and time control block management, as well as

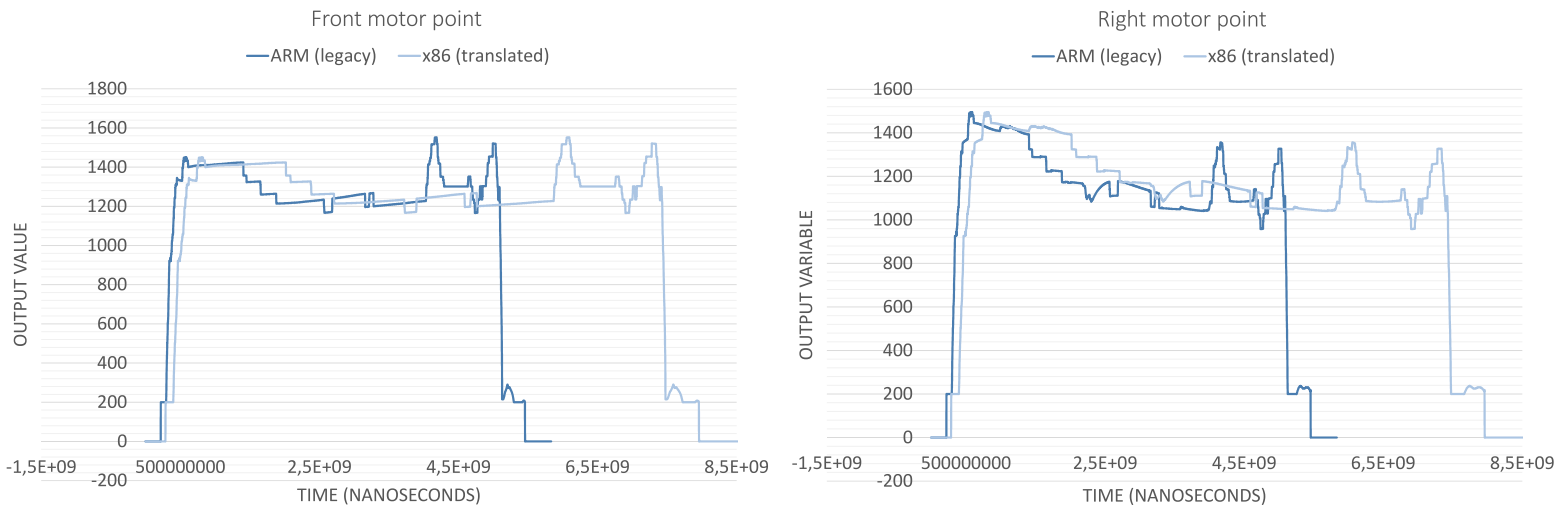


(a) Output variable 1



(b) Output variable 2

Figure 7.18.: Functional test results for the translated industrial application running on the Intel Atom E3866 processor. (a) shows for each time step (in the X-axis) the corresponding value of output variable 1 (in the Y-axis), whereas (b) shows for each time step (in the X-axis) the corresponding value of output variable 2 (in the Y-axis). Both, (a) and (b) show the results obtained for the legacy industrial application (without annotations) as well as for the annotated and then translated industrial application. On both test scenario (legacy and translated), the output value of control variable is equal, although there is a delay on the translated application with respect to the legacy application due to time control block management and static translation overhead.



(a) Front motor point

(b) Rear motor point

(c) Right motor point

(d) Left motor point

Figure 7.19.: Functional test results for the translated multirotor application running on the Intel Atom E3866 processor. (a) shows for each time step (in the X-axis) the corresponding value of motor front point variable (in the Y-axis), (b) shows for each time step (in the X-axis) the corresponding value of motor rear point variable (in the Y-axis), (c) shows for each time step (in the X-axis) the corresponding value of motor right point variable (in the Y-axis), and (d) shows for each time step (in the X-axis) the corresponding value of motor left point variable (in the Y-axis). Every graph, (a), (b), (c), and (d), show the results obtained for the legacy multirotor application (without annotations) as well as for the annotated and the translated multirotor application. On both test scenarios (legacy and translated), the output value of control variables is equal, although there is a delay on the translated application with respect to the legacy application due to time control block management and static translation overhead.

timing deviations caused by the underlying OS and hardware platform itself. For each case it might be necessary to evaluate if the introduced overhead is affordable. Once timing contracts are adjusted, as shown in Figures 7.14 to 7.16, time traces generated at runtime through the annotated time control blocks (see Tables 7.12 to 7.14) fit into the defined time contacts. Therefore, it can be concluded that the proposed timing equivalent legacy software translation incurs some overhead, but in turn it can be applied to port legacy software to a new hardware platform while preserving its timing as well as functional behaviour.

Functional test results (see Figures 7.17 to 7.19) show that the transformations applied on the timing-aware static translation process do not disturb the value observed on any of the control variable output signal. However, due to the static translation overhead, a delay is observable in the controller response time with respect to the (original) legacy system, which might disturb the functional behaviour of the overall control system. Therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the RT legacy software migration process. As described in Section 8.2, future work considers I/O virtualization in order to provides means for such analysis.

8 Conclusion and Future Work

This research work, first, reasons about the need for a portable legacy software migration solution that preserves the timing as well as the functional behaviour of the retargeted application. In the direction to cover this gap, the contributions of this thesis present a RT legacy software migration solution based on existing binary translation techniques, which are enhanced with a timing enforcement mechanism that at the same time provides means for validating the enforced timing behaviour. The proposed solution is then evaluated and as a result, this section presents the conclusions reached through this evaluation process as well as the envisioned future work to cover the limitations observed in the presented approach.

8.1 Conclusions

The analysis of the SotA answers to ***RQ1*** on how legacy software can be ported to a new architecture while preserving its timing as well as functional behaviour. As a result of the related work analysis, two machine-adaptable binary translation tools were selected to study the constraints under which is feasible to use binary translation techniques in a real-time property conserving legacy software migration process, answering to ***RQ2***. QEMU and Rev.ng are equipped with specific timing measurement support to assess the translation overhead with respect to timing. From the experimental result analysis, it can be concluded that among the proposed migration approaches, the static (based on Rev.ng) is the most appropriate method to port short-running real-time legacy code, since it ensures lower translation overhead and a more deterministic timing behaviour. Instead, the dynamic approach (based on QEMU) might be a suitable solution for porting real-time legacy code with long periods (over 0,01s) and dominated by complex floating point computations.

To find out, as stated in ***RQ3***, how expert knowledge on the legacy timing can be expressed and annotated to the legacy application, first, through a legacy system model (see definition in Section 5.1) the solution is constraint to a specific class of application. Then, based on these constraints, a set of temporal constructs are defined which provide means to systematically annotate the legacy timing behaviour onto the application. The annotated timing behaviour can then be systematically

expresses as formalized timing properties in the form of contracts. The timing enforcement assessment concludes that, given a fixed set of reference input data, the described process of lifting legacy timing properties does not disturb the functional behaviour of the legacy system. Moreover, the time traces generated at runtime by time control blocks can be used together with the formal timing specifications to perform a timing assessment on the annotated application, which concludes that a timing behaviour equivalent to that in the legacy system can be enforce through the defined temporal construct.

In order to answer ***RQ4*** and preserve the annotated timing behaviour during the migration process, the temporal constructs are implemented as a library that can be used across platforms. Though the timing-aware static legacy software translation assessment, it is proven that time traces can be generated from the translated binary on the new architecture. Moreover, answering to ***RQ5***, the timing behaviour on the new architecture is validated combining the use of time traces with formal timing specifications and MULTIC tool. Results show that although formal timing specifications needed to be relaxed such that they reflect the uncertainties generated by time control block management, the static translation process as well as the new hardware platform itself it is possible to achieve an equivalent timing behaviour on the new platform.

8.2 Future Work

Although the presented timing-aware migration approach paves the way to a RT legacy software migration, it has certain limitations, therefore, this section presents the future work that could give an answer to those limitations.

Timing-aware dynamic legacy software translation

From the feasibility study we reached to the conclusion that the static migration approach ensures a lower translation overhead and a more deterministic timing behaviour for most of the analysed cases. For this reason, the real-time legacy software migration solution has been implemented using a static translation tool. However, for some particular cases with long execution periods where code is dominated by complex floating point computations the dynamic approach turns to offer a better solution with respect to timing. Therefore, future work considers the extension of the timing-aware legacy software migration approach with a dynamic translation

alternative. This way, the most suitable approach can be chosen depending on the legacy application to be ported.

IR-level time control block annotation mechanism

The current approach presents the time control blocks which provide means to annotate the legacy timing information at source code level. As stated in ***A&C2***, this requires the legacy source code as well as the legacy toolchain to be available. This limitation could be undertaken through an IR-level timing annotation mechanism. Rev.ng translates the legacy binary into LLVM's IR, so annotations could be done at this level. However, this approach entails some other limitations, since it would not be possible to accomplish such a fine grained timing enforcement. Therefore, future work considers the implementation of such an IR-level annotation mechanism even though depending on the legacy system to be ported it might be convenient to apply a source code annotation mechanism.

System-level real-time legacy software migration

Given that the static binary translation tool does not support system-level code translation, a legacy Linux toolchain is currently required (see ***A&C3***). Before translation, the legacy application is compiled and statically linked using a Linux toolchain for the legacy architecture. To cope with this limitation it would be necessary to provide system-level code support in the static translation tool. An other alternative would be to choose the timing-aware dynamic migration solution (presented as future work also), since the dynamic binary translation tool already provides system-level code support and, therefore, there is no need for a legacy Linux toolchain.

Legacy platform dependent I/O support

As stated in ***A&C6***, the current timing-aware migration approach does not consider code that accesses legacy platform dependent I/Os, which is quite common in embedded systems. For this reason, future work should consider a process of lifting these I/O dependent code (in a similar way as it is done with timing) and implement an I/O virtualization mechanism to provide the ported legacy application with means to interact with the external environment when running on top of the new hardware platform.

Systematic adjustment of formal timing specifications

The timing test carried out either in the block-level timing enforcement assessment or in the timing-aware static legacy software translation assessment points out the need to adjust formal timing specifications such that they reflect the uncertainties generated by time control block management, the static translation process as well as the new hardware platform itself. This relaxation process is currently accomplished manually, therefore, future work considers studying the main sources that cause these timing uncertainties as well as the implementation of a systematic adjustment mechanism based on previous results.

Adjust time control blocks to overcome block management and translation overhead

The temporal constructs that control the timing based on annotations are a source of overhead as is the static translation process. For this reason, this overhead should be characterized for each particular migration process (combination of hardware platform, required time control blocks and application type) and time control blocks adjusted accordingly when possible. A possible solution would be to adapt time control blocks in such a way that they consider the overhead they incur for the deadline of the block, this way the overhead is compensated on PET and FET blocks' delay.

Extension to multi-cores

Multi-core processors play an important role in the transition to next generation embedded systems, providing concurrent resources and increased performance rates at lower clock frequencies and lower power consumption [19]. In order to maximize the resource utilization, multi-core processors share physical resources, but shared resources are a source of timing-interferences. Time contracts defined within the RT legacy software migration solution do not consider sources of interference, therefore, as stated in ***A&C1*** multi-core processors are out of the scope in this first approximation. However, future work considers the integration of multiple legacy lifted RT applications into a modern multi-core architecture with multiple other applications through a hypervisor that provides the required resource and time partitioning.

Bibliography

[1] Tesnim Abdellatif. “Rigorous Implementation of Realtime Systems”. In: (2012) (cit. on p. 27).

[2] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. “Rigorous implementation of real-time systems–from theory to application”. In: *Mathematical Structures in Computer Science* 23.4 (2013), pp. 882–914 (cit. on p. 27).

[3] *AeroSIM-RC radio control training simulator*. `http://www.aerosimrc.com/en/home.htm`. Web Page. 2019 (cit. on p. 100).

[4] *aiT: Worst-Case Execution Time Analyzer*. `http://www.absint.com/ait`. Web Page. 2005 (cit. on p. 26).

[5] Erik Altman, Bruce R Childers, Robert Cohn, et al. “08441 Final Report–Emerging Uses and Paradigms for Dynamic Binary Translation”. In: *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009 (cit. on p. 3).

[6] Rajeev Alur and David L Dill. “A theory of timed automata”. In: *Theoretical computer science* 126.2 (1994), pp. 183–235 (cit. on p. 27).

[7] Kristy Andrews and Duane Sand. “Migrating a CISC computer family onto RISC via object code translation”. In: *ACM Sigplan Notices*. Vol. 27. ACM, 1992, pp. 213–222 (cit. on p. 31).

[8] José A Baiocchi and Bruce R Childers. “Demand code paging for NAND flash in MMU-less embedded systems”. In: *Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6 (cit. on p. 35).

[9] José A Baiocchi, Bruce R Childers, Jack W Davidson, and Jason D Hiser. “Enabling dynamic binary translation in embedded systems with scratchpad memory”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.4 (2012), p. 89 (cit. on pp. 37, 39).

[10] Jose A Baiocchi and Bruce R Childers. “Heterogeneous code cache: using scratchpad and main memory in dynamic binary translators”. In: *Design Automation Conference, 2009. DAC’09. 46th ACM/IEEE*. IEEE, 2009, pp. 744–749 (cit. on p. 35).

[11] Jose A Baiocchi, Bruce R Childers, Jack W Davidson, Jason D Hiser, and Jonathan Misurda. “Fragment cache management for dynamic binary translators in embedded systems with scratchpad”. In: *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2007, pp. 75–84 (cit. on p. 35).

[12] Jose A Baiocchi, Bruce R Childers, Jack W Davidson, and Jason D Hiser. “Reducing pressure in bounded DBT code caches”. In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2008, pp. 109–118 (cit. on p. 35).

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. “Dynamo: a transparent dynamic optimization system”. In: *ACM SIGPLAN Notices* 35.5 (2000), pp. 1–12 (cit. on p. 20).

[14] Eckard Böde, Matthias Büker, Werner Damm, et al. “Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC)”. In: *FAT-Schriftenreihe 302*. 302nd ed. FAT-Schriftenreihe. Forschungsvereinigung Automobiltechnik e.V. (FAT), Oct. 2017 (cit. on pp. 13, 57).

[15] Eckard Böde, Werner Damm, Günter Ehmen, et al. “MULTIC-Tooling”. In: *FAT-Schriftenreihe 316*. 316th ed. FAT-Schriftenreihe. Forschungsvereinigung Automobiltechnik e.V. (FAT), June 2019 (cit. on pp. 13, 15, 59).

[16] Fabrice Bellard. “QEMU, a fast and portable dynamic translator”. In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46 (cit. on pp. 20, 34, 37, 39, 68, 80).

[17] Keith Bennett. “Legacy systems: Coping with success”. In: *IEEE software* 12.1 (1995), pp. 19–23 (cit. on p. 1).

[18] Arndt B Bergh, Keith Keilman, Daniel J Magenheimer, and James A Miller. “HP-3000 EMULATION ON HP PRECISION ARCHITECTURE COMPUTERS”. In: *Hewlett-Packard Journal* 38.11 (1987), pp. 87–89 (cit. on p. 31).

[19] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. “A survey of multicore processors”. In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 26–37 (cit. on p. 134).

[20] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. “Design and implementation of a dynamic optimization framework for Windows”. In: *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*. 2001 (cit. on p. 20).

[21] Eric J Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Pearson Education, 2009 (cit. on p. 25).

[22] Friederike Bruns, Philipp Ittershagen, and Kim Grüttner. “Timing Measurement and Control Blocks for Bare-Metal C++ Applications”. In: *Forum on Specification and Design Languages (FDL)*. 2019 (cit. on pp. 28–30, 39, 45, 50, 67, 155).

[23] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001 (cit. on p. 25).

[24] Paul M Cashman and Anatol W Holt. “A communication-oriented approach to structuring the software maintenance environment”. In: *ACM SIGSOFT Software Engineering Notes* 5.1 (1980), pp. 4–17 (cit. on p. 1).

[25] Jiunn-Yeu Chen, Wuu Yang, Tzu-Han Hung, Hong-Men Su, and Wei-Chung Hsu. “A static binary translator for efficient migration of ARM-based applications”. In: *Workshop on Optimizations for DSP and Embedded Systems*. Citeseer, 2008 (cit. on pp. 36–39).

[26]Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. “Mojo: A Dynamic Optimization System”. In: *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)* (2000), pp. 81–90 (cit. on p. 20).

[27]René J Chevance. *Server architectures: Multiprocessors, clusters, parallel systems, web servers, storage solutions*. Digital Press, 2004 (cit. on p. 2).

[28]*Chrono in C++*. `https://www.geeksforgeeks.org/chrono-in-c/`. Web Page. 2019 (cit. on p. 73).

[29]C. Cifuentes and M. Van Emmerik. “UQBT: adaptable binary translation at low cost”. In: *Computer* 33.3 (2000), pp. 60–66 (cit. on pp. 3, 33, 36, 37, 39).

[30]C. Cifuentes and V. Malhotra. “Binary translation: static, dynamic, retargetable?” In: *1996 Proceedings of International Conference on Software Maintenance*. 1996, pp. 340–349 (cit. on p. 31).

[31]Alessandro Cimatti and Stefano Tonetta. “Contracts-refinement proof system for component-based embedded systems”. In: *Science of computer programming* 97 (2015), pp. 333–348 (cit. on p. 59).

[32]Bob Cmelik and David Keppel. “Shade: A fast instruction-set simulator for execution profiling”. In: *Fast Simulation of Computer Architectures*. Springer, 1995, pp. 5–46 (cit. on p. 31).

[33]Bryce Cogswell and Zary Segall. “Timing insensitive binary to binary translation of real time systems”. In: *Workshop on Architectures for Real-Time Applications, ISCA*. 1995 (cit. on pp. 3, 31, 32, 37–39).

[34]James C Dehnert, Brian K Grant, John P Banning, et al. “The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2003, pp. 15–24 (cit. on p. 20).

[35]Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A Fisher. “Deli: A new run-time control point”. In: *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 2002, pp. 257–268 (cit. on p. 20).

[36]Premkumar T Devanbu. “Re-targetability in software tools”. In: *ACM SIGAPP Applied Computing Review* 7.3 (1999), pp. 19–26 (cit. on p. 29).

[37]Len Erlikh. “Leveraging legacy system dollars for e-business”. In: *IT professional* 2.3 (2000), pp. 17–23 (cit. on p. 1).

[38]Maher Fakih, Kim Grüttner, Sören Schreiner, et al. “Experimental Evaluation of SAFEPOWER Architecture for Safe and Power-Efficient Mixed-Criticality Systems”. In: *Journal of Low Power Electronics and Applications* 9.1 (2019), p. 12 (cit. on p. 91).

[39]Heiko Falk and Paul Lokuciejewski. “A compiler framework for the reduction of worst-case execution times”. In: *Real-Time Systems* 46.2 (2010), pp. 251–300 (cit. on pp. 26, 29–31, 39, 94).

[40] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. “rev.ng: a unified binary analysis framework to recover CFGs and function boundaries”. In: *Proceedings of the 26th International Conference on Compiler Construction*. 3033028: ACM, 2017, pp. 131–141 (cit. on pp. 34, 37, 39, 82).

[41] C. J. Fidge. “Formal change impact analyses for emulated control software”. In: *International Journal on Software Tools for Technology Transfer* 8.4 (2006), pp. 321–335 (cit. on p. 26).

[42] Narain Gehani and Krithi Ramamritham. “Real-time Concurrent C: A language for programming dynamic real-time systems”. In: *Real-Time Systems* 3.4 (1991), pp. 377–405 (cit. on pp. 28, 30, 39).

[43] Tayfun Gezgin, Raphael Weber, and Markus Oertel. “Multi-aspect virtual integration approach for real-time and safety properties”. In: *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS14)*. 2014 (cit. on p. 59).

[44] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. “Memory optimization of dynamic binary translators for embedded systems”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.3 (2012), p. 22 (cit. on pp. 35, 37, 39).

[45] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. “Reducing exit stub memory consumption in code caches”. In: *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 2007, pp. 87–101 (cit. on p. 35).

[46] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. “The Mälardalen WCET Benchmarks – Past, Present and Future”. In: *WCET2010*. Ed. by Björn Lisper. Brussels, Belgium: OCG, July 2010, pp. 137–147 (cit. on pp. 89, 93).

[47] Kim Hazelwood and Artur Klauser. “A dynamic binary instrumentation engine for the ARM architecture”. In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 261–270 (cit. on p. 20).

[48] Thomas Heinz. “Preserving temporal behaviour of legacy real-time software across static binary translation”. In: *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. ACM, 2008, pp. 1–4 (cit. on pp. 3, 31, 32, 37–39).

[49] Thomas A Henzinger and Christoph M Kirsch. “The Embedded Machine: Predictable, portable real-time code”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.6 (2007), p. 33 (cit. on p. 26).

[50] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. “Giotto: A time-triggered language for embedded programming”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99 (cit. on p. 25).

[51] *Homepage of the Accellera Systems Initiative*. `http://www.accellera.org`. Web Page. 2019 (cit. on p. 60).

[52] Ray Hookway. “DIGITAL FX! 32 running 32-Bit x86 applications on Alpha NT”. In: *Compcon’97. Proceedings, IEEE*. IEEE, 1997, pp. 37–42 (cit. on p. 31).

[53] Wei Hu, Jason Hiser, Dan Williams, et al. “Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation”. In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE ’06. ACM, 2006, pp. 2–12 (cit. on p. 20).

[54] “Chapter 2 - Virtualization”. In: *Mobile Cloud Computing*. Ed. by Dijiang Huang and Huijun Wu. Morgan Kaufmann, 2018, pp. 31 –64 (cit. on p. 9).

[55] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. “DisIRer: Converting a retargetable compiler into a multiplatform binary translator”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 7.4 (2010), p. 18 (cit. on pp. 34, 37, 39).

[56] *Imperas Ltd. Open Virtual Platforms (OVP)*. `http://www.ovpworld.org/`. Web Page. 2018 (cit. on p. 20).

[57] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012 (cit. on p. 73).

[58] Inkyu Kim. “Timing analysis in binary-to-binary translation”. In: (1998) (cit. on p. 11).

[59] Vladimir Kiriansky, Derek Bruening, and Saman P Amarasinghe. “Secure Execution via Program Shepherding”. In: *USENIX Security Symposium*. Vol. 92. 2002, p. 84 (cit. on p. 20).

[60] Christoph M Kirsch and Ana Sokolova. “The logical execution time paradigm”. In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120 (cit. on p. 25).

[61] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science + Business Media, 2011 (cit. on pp. 10, 12).

[62] Hermann Kopetz and Wilhelm Ochsenreiter. “Clock synchronization in distributed real-time systems”. In: *IEEE Transactions on Computers* 100.8 (1987), pp. 933–940 (cit. on p. 11).

[63] Phillip A Laplante and Seppo J Ovaska. *Real-time systems design and analysis: tools for the practitioner*. John Wiley and Sons, 2011 (cit. on p. 25).

[64] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75 (cit. on p. 34).

[65] Briag Le Nabec, Belgacem Ben Hedia, Jean-Philippe Babau, Mathieu Jan, and Hela Guesmi. “Modeling legacy code with BIP: how to reduce the gap between formal description and real-time implementation”. In: *2016 Forum on Specification and Design Languages (FDL)*. IEEE, 2016, pp. 1–8 (cit. on pp. 27, 29, 30, 39).

[66] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. “Design and implementation of a lightweight dynamic optimization system”. In: *Journal of Instruction-Level Parallelism* 6.4 (2004), pp. 332–341 (cit. on p. 20).

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices*. Vol. 40. ACM, 2005, pp. 190–200 (cit. on pp. 20, 35, 37).

[68] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, et al. “DBILL: an efficient and retargetable dynamic binary instrumentation framework using llvm backend”. In: *Acm Sigplan Notices*. Vol. 49. ACM, 2014, pp. 141–152 (cit. on p. 20).

[69] Cathy May. *Mimic: a fast system/370 simulator*. Vol. 22. ACM, 1987 (cit. on p. 31).

[70] James R McKee. “Maintenance as a function of design”. In: *Proceedings of the July 9-12, 1984, national computer conference and exposition*. ACM, 1984, pp. 187–193 (cit. on p. 1).

[71] Harlan D Mills. “Software development”. In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 265–273 (cit. on p. 1).

[72] Victor Moya. “Study of the techniques for emulation programming”. In: *Proyecto fin de carrera. Universidad Politécnica de Cataluña. España* (2001) (cit. on p. 21).

[73] Saranya Natarajan and David Broman. “Timed C: An Extension to the C Programming Language for Real-Time Systems”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 227–239 (cit. on pp. 27–30, 39).

[74] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *SIGPLAN Not.* 42.6 (2007), pp. 89–100 (cit. on p. 20).

[75] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012 (cit. on pp. 7, 8).

[76] John T Nosek and Prashant Palvia. “Software maintenance management: changes in the last decade”. In: *Journal of Software Maintenance: Research and Practice* 2.3 (1990), pp. 157–174 (cit. on p. 1).

[77] *OFFIS multirotor*. `https://multirotor.offis.de/wordpress/`. Web Page. 2019 (cit. on p. 91).

[78] J. Perez, A. Perez, and R. Obermaisser. “Executable Time-Triggered Model (E-TTM) for Real-Time Control Systems”. In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2010, pp. 42–49 (cit. on p. 11).

[79] Chittoor V Ramamoorthy, Atul Prakash, Wei Tek Tsai, and Yutaka Usuda. “Software engineering: Problems and perspectives”. In: *Computer* 17.10 (1984), pp. 191–209 (cit. on p. 1).

[80] Stefan Resmerita, Andreas Naderlinger, Manuel Huber, Kenneth Butts, and Wolfgang Pree. “Applying real-time programming to legacy embedded control software”. In: *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 2015, pp. 1–8 (cit. on pp. 25, 29, 30, 39).

[81] John Reutter III. “Maintenance is a management problem and a programmer’s opportunity”. In: *Proceedings of the May 4-7, 1981, national computer conference*. ACM, 1981, pp. 343–347 (cit. on p. 1).

[82] Consortium SAFEPOWER. “D4.6 Final cross-domain public demonstrator”. In: (2017) (cit. on p. 92).

[83] Kevin Scott, Jack W Davidson, and Kevin Skadron. “Low-overhead software dynamic translation”. In: *University of Virginia, Charlottesville, VA* (2001) (cit. on p. 20).

[84] Kevin Scott, Naveen Kumar, Siva Velusamy, et al. “Retargetable and reconfigurable software dynamic translation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2003, pp. 36–47 (cit. on pp. 35, 37).

[85] Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu. “An LLVM-based hybrid binary translation system”. In: *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*. IEEE, 2012, pp. 229–236 (cit. on pp. 22, 39, 40).

[86] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. “LLBT: an LLVM-based static binary translator”. In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 51–60 (cit. on pp. 3, 36–39).

[87] G. M. Silberman and K. Ebcioglu. “An architectural framework for supporting heterogeneous instruction-set architectures”. In: *Computer* 26.6 (1993), pp. 39–56 (cit. on p. 31).

[88] Richard L Sites, Anton Chernoff, Matthew B Kirk, Maurice P Marks, and Scott G Robinson. “Binary translation”. In: *Communications of the ACM* 36.2 (1993), pp. 69–81 (cit. on p. 31).

[89] Josef Templ. “Timing definition language (TDL) 1.5 specification”. In: *University of Salzburg, Tech. Rep* 24 (2007) (cit. on p. 26).

[90] Scott R Tilley and Dennis Smith. *Perspectives on legacy system reengineering*. 1995 (cit. on p. 2).

[91] David Ung and Cristina Cifuentes. “Machine-adaptable dynamic binary translation”. In: *ACM SIGPLAN Notices*. Vol. 35. ACM, 2000, pp. 41–51 (cit. on pp. 33, 37, 39).

[92] Hans Van Vliet. *Software engineering: principles and practice*. Vol. 13. Wiley, 2008 (cit. on p. 1).

[93] Christian Wagner and Christian Wagner. *Model-Driven Software Migration*. Springer, 2014 (cit. on p. 1).

[94] M. Wahler, R. Eidenbenz, C. Franke, and Y. A. Pignolet. “Migrating legacy control software to multi-core hardware”. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. 2015, pp. 458–466 (cit. on p. 1).

[95] Andrew Wellings. *Concurrent and real-time programming in Java*. John Wiley & Sons, Inc., 2004 (cit. on p. 25).

[96] Andy Wellings and Alan Burns. “Real-time utilities for Ada 2005”. In: *International Conference on Reliable Software Technologies*. Springer. 2007, pp. 1–14 (cit. on p. 25).

[97] Bing Wu, Deirdre Lawless, Jesus Bisbal, et al. “Legacy system migration: A legacy data migration engine”. In: *Proceedings of the 17th International Database Conference (DATASEM’97)*. 1997, pp. 129–138 (cit. on p. 1).

[98]Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, and Bo Liu. *Crossbit: a multi-sources and multi-targets DBT*. Journal Article. 2010 (cit. on pp. 3, 34, 37, 39).

List of Figures

List of Tables

List of Listings

Acronyms

ABI Application Binary Interface

ADC Analog-to-Digital Converter

API Application Programming Interface

AST Abstract Syntax Tree

ASTRA Automated Synthesis of TRAnslators

BB Basic-Block

BET Budgeted Execution Time

BIP Behaviour Interaction Priority

BNF Backus-Naur Form

BSP Board Support Package

BT Binary Translation

CERTS Critical Embedded Real-Time Systems

CFG Control Flow Graph

CISC Complex Instruction Set Computer

CPI Cycles Per Instruction

CPU Central Processing Unit

DBT Dynamic Binary Translation

EB Evaluation Board

EET Estimated Execution Time

E-machine Embedded Machine

FET Forced Execution Time

FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

GCD Greatest Common Divisor

HAL Hardware Abstraction Layer

HRTL Higher-Level Register Transfer Language

IR Intermediate Representation

ISA Instruction Set Architecture

I/O Input/Output

LCM Least Common Multiple

LET Logical Execution Time

LLVM Low Level Virtual Machine

MTSL MULTIC Time Specification Language

N/R Not Relevant

OS Operating System

PC Program Counter

PET Periodic Execution Time

PNET Period N Execution Time

POSIX Portable Operating System Interface

QEMU Quick EMUlator

RISC Reduced Instruction Set Computer

RT Real-Time

RTL Register-Transfer Level

RT-BIP Real-Time BIP

RT-BIPAgent Real-Time BIP Agent

SBT Static Binary Translation

SoC System on a Chip

SotA State of The Art

SPM Scratchpad Memory

TB Translation Block

TCG Tiny Code Generator

TDL Timing Definition Language

TMCB Timing Measurement and Control Block

VIT Virtual Integration Test

VM Virtual Machine

WCC WCET-aware C Compiler

WCEP Worst Case Execution Path

WCET Worst Case Execution Time

A Systematic Annotation & Transformation

This appendix presents the algorithms for the systematic annotation of legacy code using the time measurement and control blocks, as well as the systematic transformation of annotated legacy code into formal timing specification for the latter validation of the timing behaviour on the new architecture.

A.1 Systematic Annotation with Time Measurement Blocks

The profiling phase in the RT legacy software migration flow, presented in Section 5.2.1, describes how the time measurement block, EET [22], can be used to easily perform execution time measurements on the legacy code. Timing measurements are then evaluated by an expert, together with code analysis and legacy system's timing specifications, to extract the necessary timing information from the legacy system.

In order to systematically annotate the legacy code with time measurement blocks, algorithm A.1.1 sorts out the legacy task set T (see definition 5.1.1) according to the execution model (see definition 5.1.2), resulting in a sorted task set T_{SO}:

Definition A.1.1. (Sorted Task Set). T_{SO} consists of a set of sorted tasks, where each task is represented by a tuple $(p_i, \phi_i, e_i, d_i, cr_i)$ (see Definition 5.1.1 for the description of each element in the tuple).

Algorithm A.1.1 Sort out task set.

Input: Set of time slots $sl_{j,1}, sl_{j,2}, \ldots, sl_{j,n}$.
Set of frames $\{f_j\}$.
Output: Set of sorted tasks T_{SO}.

1 $i, j, k \leftarrow 0$ ▷ Initialize task, frame and slot identifiers to 0
2 $cntSL \leftarrow 0$ ▷ Initialize frame and slot counters to 0

```
3  T_SO ← {NULL}                                   ▷ Initialize sorted task set to NULL
4  for all f_j ∈ {f_j} do
5      cntF ← cntF + 1
6  end for
7  for all sl_{1,k} ∈ f_1 do
8      cntSL ← cntSL + 1
9  end for
10 while k < cntSL do
11     while j < cntF do
12         if t_i ≠ sl_{j,k}.t then
13             t_i ← sl_{j,k}.t
14             i ← i + 1
15         end if
16         j ← j + 1
17     end while
18     k ← k + 1
19 end while
```

Then, the sorted task set T_{SO} is annotated with time measurement blocks following algorithms A.1.2 to A.1.4. The output of the main algorithm A.1.2 consist of a set of time measurement block nodes MB, sorted from left to right, starting from the root node and incrementing the level when there is no node left in the current level:

Definition A.1.2. (Time Measurement Block Tree). Each node in the time measurement block node set $MB = \{mb_m\}$ consists of a tuple (nm, l, p, tf), where nm is the name of the node that is composed of the type tp (EET) and an identifier id ($id \in \mathbb{N}$), l represents the level of the node, p is the parent node and tf consists of the task functions that are wrapped by the time measurement block of each node.

Algorithm A.1.2 Systematic annotation of legacy code with time measurement blocks.

Input: Set of sorted tasks T_{SO}.
Frame size F.
Output: Set of time measurement block nodes MB.

```
1  i, m ← 1                                        ▷ Initialize task and node identifiers to 0
2  l ← 1                                           ▷ Initialize node level to 1
3  p ← NULL                                        ▷ Initialize node parent to NULL
4  tf ← t_0                                        ▷ Initialize node task function to t_0
5  n ← 0                                           ▷ Initialize number of tasks to 0
6  crCnt, crMax ← 0                                ▷ Initialize critical task counter and maximum to 0
7  cntAp, Ap ← 0                                   ▷ Initialize active period task counter and active period to 0
```

```
8  cnt ← 0                                                    ▷ Initialize counter to 0
9  for all t_i ∈ T_SO do
10     tf ← tf", "t_i
11     n ← n + 1
12     if t_i.cr = 1 then
13         crMax ← crMax + 1
14     end if
15 end for
16 (MB) ← EETNode(m, l, p, tf)
17 l ← l + 1
18 m ← m + 1
19 p ← 0
20 tf ← t_0
21 while i ≤ n do
22     if crMax − crCnt > 1 or (crMax − crCnt = 1 and t_i.cr ≠ 1) then
23         repeat
24             (Ap, cntAp, crCnt) ← TaskManagement(T_SO, F, i, cntAp, crCnt)
25             tf ← tf", "t_i
26             i ← i + 1
27         until t_i.cr = 1 and (t_i.p/F ≠ Ap or cntAp = Ap)
28     else
29         tf ← t_i
30         i ← i + 1
31     end if
32     (MB) ← EETNode(m, l, p, tf)
33     l ← l + 1
34     m ← m + 1
35     Ap ← 0
36     cntAp ← 0
37 end while
38 cnt ← m − 1
39 i ← 0
40 while i ≤ n and (crMax − crCnt > 1 or (crMax − crCnt = 1 and t_i.cr ≠ 1)) do
41     repeat
42         (Ap, cntAp, crCnt) ← TaskManagement(T_SO, F, i, cntAp, crCnt)
43         p ← m − cnt
44         tf ← t_i
45         (MB) ← EETNode(m, l, p, tf)
46         m ← m + 1
47         cnt ← cnt + 1
48     until t_i.cr = 1 and (t_i.p/F ≠ Ap or cntAp = Ap)
49     cnt ← cnt − 1
50     Ap ← 0
51     cntAp ← 0
```

```
52 end while
```

Algorithm A.1.3 Manage tasks with period greater than F and task containing critical sections.

Input: Set of sorted tasks T_{SO}.
Frame size F.
Task identifier i
Active period task counter $cntAp$
Critical task counter $crCnt$
Output: Task's active period Ap.
Updated active period task counter $cntAp$.
Updated critical section counter $crCnt$.

```
1  procedure TaskManagement(T_SO, F, i, cntAp, crCnt)
2      if t_i.p > F then
3          if cntAp = 0 then
4              Ap ← t_i.p/F
5          end if
6          cntAp ← cntAp + 1
7      end if
8      if t_i.cr = 1 then
9          crCnt ← crCnt + 1
10     end if
11 end procedure
```

Algorithm A.1.4 Create EET node.

Input: Node identifier m.
Node level l.
Node parent p
Task functions wrapped by the node tf
Output: Set of time measurement block nodes MB.

```
1  procedure TaskManagement(m, l, p, tf)
2      mb_m.nm.tp ← "EET"
3      mb_m.nm.id ← k
4      mb_m.l ← l
5      mb_m.p ← p
6      mb_m.tf ← tf
7  end procedure
```

A.2 Systematic Annotation with Time Control Blocks

From the profiling phase, legacy timing properties and legacy behaivoural code are obtained. Then, using algorithms A.2.1 to A.2.4 and A.3.1, the behavioural legacy code is annotated with time control blocks to make the legacy timing behaviour explicit. The input to the main algorithm (algorithm A.2.1) consists of the frame size F and a set of sorted tasks where every time element has the same time unit $T_{SO_{unit}}$:

Definition A.2.1. (Unit Transformation Function). Function $\beta : T_{SO} \rightarrow T_{SO_{unit}}$ transforms every time element in the sorted task set T_{SO} to sorted elements with the same time unit, resulting in $T_{SO_{unit}}$.

The output resulting from Algorithm A.2.1 consists of a set of time control block nodes CB, where nodes are sorted from left to right, starting from the root node and incrementing the level when there is no any node left at the current level:

Definition A.2.2. (Time Control Block Tree). Each node cb_m in the time control block node set CB consists of a tuple $(nm, l, p, bg, of, tu, ap, op, tf)$, where nm is the name of the node that is composed of the type tp (be it PET, FET, BET or PNET) and an identifier id ($id \in \mathbb{N}$), l represents the level of the node, p is the parent node, bg is the budget assigned to the node, of is the offset, tu is the time unit for the bg, ap and op are the active period and the offset in periods, which is only necessary in nodes with tp_i=PNET, whereas tf consists of the task functions that are wrapped by the control block this node corresponds to.

Algorithm A.2.1 Systematic annotation of legacy code with time control blocks.

Input: Set of sorted tasks $T_{SO_{unit}}$.
Frame size F.
Output: Set of time control block nodes CB.

1 $i, m \leftarrow 1$ ▷ Initialize task and node identifiers to 0
2 $l \leftarrow 1$ ▷ Initialize node level to 1
3 $p \leftarrow 0$ ▷ Initialize node parent to root node
4 $n \leftarrow 0$ ▷ Initialize number of tasks to 0
5 $crCnt, crMax \leftarrow 0$ ▷ Initialize critical task counter and maximum to 0
6 $pnetOp, pnetAp \leftarrow 0$ ▷ Initialize PNET offset and active period to 0
7 $fetCnt, fetOf \leftarrow 0$ ▷ Initialize FET counter and offset to 0
8 **for all** $t_i \in \{t_i\}$ **do**
9 $n \leftarrow n + 1$

```
    if t_i.cr = 1 then
        crMax ← crMax + 1
    end if
end for
(CB) ← PETNode(T_SO_unit, i, m, l, CB)
m ← m + 1
l ← l + 1
while i ≤ n do
    if crMax − crCnt > 1 or (crMax − crCnt = 1 and t_i.cr ≠ 1) then
        (CB, i, m, crCnt, pnetOp, pnetAp, fetOf) ← FETNode(T_SO_unit,
            F, i, m, l, CB, pnetOp, pnetAp, fetOf, crCnt)
    else
        (CB, pnetOp, pnetAp, crCnt) ← BETPNETNode(T_SO_unit, F, i, m,
            l, p, pnetOp, pnetAp, crCnt)
        if t_i.p/F ≠ pnetAp or pnetOp = pnetAp or t_i.p = F then
            pnetAp ← 0
            pnetOp ← 0
        end if
    end if
end while
fetCnt ← m − 1
i ← 0
l ← l + 1
while i ≤ n and (crMax − crCnt > 1 or (crMax − crCnt = 1 and t_i.cr ≠ 1)) do
    repeat
        p ← m − fetCnt
        (CB, pnetOp, pnetAp, crCnt) ← BETPNETNode(T_SO_unit, F, i, m,
            l, p, pnetOp, pnetAp, crCnt)
        fetCnt ← fetCnt + 1
    until t_i.cr = 1 and (t_i.p/F ≠ pnetAp or pnetOp = pnetAp or t_i.p = F)
    fetCnt ← fetCnt − 1
    pnetAp ← 0
    pnetOp ← 0
end while
```

Algorithm A.2.2 Create root PET node.

Input: Set of sorted tasks $T_{SO_{unit}}$.
Task and node identifiers i, m.
Tree node level l.
Set of control block nodes CB.

Output: Updated set of control block nodes CB.

```
1 procedure PETNode(T_SO_unit, i, m, l, CB)
```

```
2     cb_m.nm.tp ← "PET"
3     cb_m.nm.id ← m
4     cb_m.l ← l
5     cb_m.bg ← F.v
6     cb_m.of ← t_i.φ.v
7     cb_m.tu ← t_i.p.tu
8     for all t_i ∈ {t_i} do
9         cb_m.tf ← cb_m.tf", "t_i
10    end for
11 end procedure
```

Algorithm A.2.3 Create FET node.

Input: Set of sorted tasks $T_{SO_{unit}}$.
Frame size F.
Task and node identifiers i, m.
Tree node level l.
Set of control block nodes CB.
PNET offset and active period $pnetOp, pnetAp$.
FET node offset $fetOf$.

Output: Updated set of control block nodes CB.
Updated critical section counter $crCnt$.
Updated PNET offset and active period $pnetOp, pnetAp$.
Updated FET node offset $fetOf$.

```
1  procedure FETNODE(T_SO_unit, F, i, m, l, CB, pnetOp, pnetAp, fetOf)
2     cb_m.nm.tp ← "FET"
3     cb_m.nm.id ← m
4     cb_m.l ← l
5     cb_m.p ← 0
6     cb_m.of ← t_i.φ.v
7     cb_m.tu ← t_i.p.tu
8     repeat
9        if t_i.p > F then
10          if pnetOp = 0 then
11             pnetAp ← t_i.p/F
12          end if
13          pnetOp ← pnetOp + 1
14       end if
15       if t_i.cr = 1 then
16          crCnt ← crCnt + 1
17       end if
```

```
18          cb_m.bg ← t_{i+1}.φ.v − fetOf
19          cb_m.tf ← cb_m.tf", "t_i
20          i ← i + 1
21      until t_i.cr = 1 and (t_i.p/F ≠ pnetAp or pnetOp = pnetAp or t_i.p = F)
22      m ← m + 1
23      pnetAp ← 0
24      pnetOp ← 0
25      fetOf ← fetOf + cb_{m−1}.bg
26  end procedure
```

Algorithm A.2.4 Create BET/PNET node.

Input: Set of sorted tasks $T_{SO_{unit}}$.
Frame size F.
Task and node identifiers i, m.
Tree node level and parent l, p.
Critical section counter $crCnt$.
PNET offset and active period $pnetOp, pnetAp$.
Output: Updated set of control block nodes CB.
Updated critical section counter $crCnt$.
Updated PNET offset and active period $pnetOp, pnetAp$.

```
1  procedure BETPNETNODE(T_{SO_unit}, F, i, m, l, p, crCnt, pnetOp, pnetAp)
2      if t_i.p > F then
3          cb_m.nm.tp ← "PNET"
4          if pnetOp = 0 then
5              pnetAp ← t_i.p/F
6          end if
7          cb_m.ap ← pnetAp
8          cb_m op ← pnetOp
9          pnetOp ← pnetOp + 1
10     else
11         cb_m.nm.tp ← "BET"
12     end if
13     cb_m.nm.id ← m
14     cb_m.l ← l
15     cb_m.p ← p
16     cb_m.bg ← t_i.e.v
17     cb_m.of ← t_i.φ.v
18     cb_m.tu ← t_i.p.tu
19     cb_m.tf ← t_i
20     if t_i.cr = 1 then
```

21 $crCnt \leftarrow crCnt + 1$
22 **end if**
23 $i \leftarrow i + 1$
24 $m \leftarrow m + 1$
25 **end procedure**

A.3 Systematic Transformation to Formal Timing Specifications

In order to verify the correct timing behaviour of annotated legacy code, legacy timing properties are systematically transformed into formal timing specifications (formal timing specifications are based on the MTSL described in section 2.3). Algorithms A.3.1 to A.3.5 describe the systematic transformation of the time control block annotated legacy code into a component-contract structure. The input to the main algorithm (see Algorithm A.3.1) consist of a set of control block nodes $CB = \{cb_m\}$, as described in definition A.2.2, whereas the output consists of a set of component nodes $CO = \{co_m\}$ with their corresponding contract in the form of assumptions A and guarantees G:

Definition A.3.1. (Component Tree). CO is a set of component nodes where each node consists of a tuple $(nm, l, p, EN, ex, A, G, tf)$, where nm is the name of the node that is composed of the type tp (be it PET, FET, BET or PNET) and an identifier id ($id \in \mathbb{N}$), p is the parent node, EN is a set of entry ports, ex is the exit port, A is a set of assumptions, G is a set of guarantees and tf consists of the task functions that correspond to the component node. Each entry and exit port consists of a port name pn and a port connexion pc, which determines the output/input port it is connected to.

Algorithm A.3.1 Transforming timing annotations into timing specifications.

Input: Set of control block nodes CB.
Output: Set of component nodes CO.

1 $\forall m : co_m.nm \leftarrow cb_m.nm$ ▷ Every component node inherits the name from the control block node
2 $\forall m : co_m.l \leftarrow cb_m.l$ ▷ Every component node inherits the level from the control block node
3 $\forall m : co_m.p \leftarrow cb_m.p$ ▷ Every component node inherits the parent from the control block node

```
4  ∀m : co_m.tf ← cb_m.tf      ▷ Every component node inherits the task functions from the
   control block node
5  m ← 0                                                    ▷ Start with the first node
6  F ← cb_m.bg                  ▷ Set the size of the frame to the rood node (PET) budget
7  Bg ← 0                       ▷ Budget to be passed within sibling BET and PNET nodes
8  Fst ← 1                            ▷ Flag to identify the first sibling BET/PNET node
9  cntPNET, cntBET ← 0          ▷ Initialize sibling PNET and BET node counters to 0
10 for all cb_m ∈ CB do
11     if cb_m.nm.tp = PET then
12         (CO, m) ← PETTransformation(CB, m, F)
13     else if cb_m.nm.tp = FET then
14         (CO, m, Fst, cntPNET) ← FETTransformation(CB, m, F, Bg)
15     else if cb_m.nm.tp = BET then
16         (CO, m, Bg, Fst, cntPNET, cntBET) ← BETTransformation(CB, m, F,
               Bg, Fst, cntBET)
17     else if cb_m.nm.tp = PNET then
18         (CO, m, Bg, Fst, cntPNET, cntBET) ← PNETTransformation(CB, m, F,
               Bg, Fst, cntPNET, cntBET)
19     end if
20 end for
```

Algorithm A.3.2 PET block transformation.

Input: Set of control block nodes CB.
Node identifier m.
Frame size F.
Output: Set of component nodes CO.
Updated node identifier m.

```
1  procedure PETTRANSFORMATION(CB, m, F)
2      co_m.en_0.pn ← Entry
3      co_m.ex.pn ← Exit
4      co_m.a_0 ← co_m.en_0.pn" occurs every ["F", "F"] "cb_m.tu" with
           offset ["(cb_m.of)^+", "cb_m.of"] "cb_m.tu"."
5      co_m.g_0 ← "Reaction ("co_m.en_0.pn", "co_m.ex.pn") within ["F", "F
           "] "cb_m.tu"."
6      m ← m + 1
7  end procedure
```

Algorithm A.3.3 FET block transformation.

Input: Set of control block nodes CB.
Node identifier m.

Frame size F.

Output: Set of component nodes CO.
Updated node identifier m.

```
1  procedure FETTRANSFORMATION(CB, m, F)
2      co_m.en_0.pn ← Entry
3      co_m.ex.pn ← Exit
4      if co_m.p ≠ co_{m−1}.p then
5          co_m.en_0.pc ← co_{co_m.p}.en_0
6      else
7          co_m.en_0.pc ← co_{m−1}.ex
8      end if
9      co_m.a_0 ← co_m.en_0.pn" occurs every ["F", "F"] "cb_m.tu" with offset ["cb_m.of", "
           cb_m.of"] "cb_m.tu"."
10     co_m.g_0 ← "Reaction ("co_m.en_0.pn", "co_m.ex.pn") within ["cb_m.bg", "cb_m.bg"] "
           cb_m.tu"."
11     m ← m + 1
12 end procedure
```

Algorithm A.3.4 BET block transformation.

Input: Set of control block nodes CB.
Node identifier m.
Frame size F.
Budget Bg.
First sibling BET/PNET node flag Fst.
BET block counter $cntBET$.

Output: Set of component nodes CO.
Updated node identifier m.
Updated budget Bg.
Updated first sibling BET/PNET node flag Fst.
Updated PNET block counter $cntPNET$.
Updated BET block counter $cntBET$.

```
1  procedure BETTRANSFORMATION(CB, m, F, Bg, Fst, cntBET)
2      co_m.en_0.pn ← Entry
3      co_m.ex.pn ← Exit
4      if co_m.p ≠ co_{m−1}.p then
5          co_m.en_0.pc ← co_{co_m.p}.en_0
6          co_{m−1}.ex.pc ← co_{m−1}.ex.pc', 'co_{co_{m−1}.p}.ex.pc
7          Bg ← 0
8          Fst ← 1
9          cntPNET ← 0
```

```
10        cntBET ← 0
11    else
12        co_m.en_0.pc ← co_{m−1}.ex
13        if co_{m−1}.nm.tp = PNET then
14            Fst ← 0
15        end if
16    end if
17    if Fst then
18        co_m.a_0 ← co_m.en_0.pn" occurs every ["F", "F"] "cb_m.tu" with offset ["cb_m.of", "
              cb_m.of"] "cb_m.tu"."
19        co_m.g_0 ← "Reaction ("co_m.en_0.pn", "co_m.ex.pn") within [0, "cb_m.bg"] "
              cb_m.tu"."
20        Fst ← 0
21    else
22        co_m.en_1.pn ← PEntry
23        co_m.en_1.pc ← co_{co_m.p}.en_0
24        co_m.a_0 ← co_m.en_0.pn" occurs every ["F − Bg", "F + Bg"] "cb_m.tu" with offset"
              " ["(cb_m.of − Bg)", "cb_m.of"] "cb_m.tu"."
25        co_m.a_1 ← co_m.en_1.pn" occurs every ["F", "F"] "cb_m.tu" with offset ["
              cb_{cb_m.p}.of", "cb_{cb_m.p}.of"] "cb_{cb_m.p}.tu"."
26        co_m.g_0 ← "Reaction ("co_m.en_0.pn", "co_m.ex.pn") within [0, "cb_m.bg + Bg"] "
              cb_m.tu"."
27        if co_{co_m.p}.nm.tp = PET then
28            co_m.g_1 ← "Reaction ("co_m.en_1.pn", "co_m.ex.pn") within ["(cb_m.of − Bg
                  )", "cb_m.of + cb_m.bg"] "cb_m.tu"."
29        else
30            co_m.g_1 ← "Reaction ("co_m.en_1.pn", "co_m.ex.pn") within [0, "cb_m.bg + Bg
                  "] "cb_m.tu"."
31        end if
32    end if
33    cntBET ← cntBET + 1
34    m ← m + 1
35    Bg ← Bg + cb_m.bg
36 end procedure
```

Algorithm A.3.5 PNET block transformation.

Input: Set of control block nodes CB.
Node identifier m.
Frame size F.
Budget Bg.
First sibling BET/PNET node flag Fst.
PNET block counter $cntPNET$.
BET block counter $cntBET$.

Output: Set of component nodes CO.
Updated node identifier m.
Updated budget Bg.
Updated first sibling BET/PNET node flag Fst.
Updated PNET block counter $cntPNET$.
Updated BET block counter $cntBET$.

1 **procedure** PNETTRANSFORMATION($CB, m, F, Bg, Fst, cntPNET, cntBET$)
2 $co_m.en_0.pn \leftarrow Entry$
3 $co_m.ex.pn \leftarrow Exit$
4 **if** $cb_m.p \neq cb_{m-1}.p$ **then**
5 $co_m.en_0.pc \leftarrow co_{co_m.p}.en_0$
6 $co_{m-1}.ex.pc \leftarrow co_{m-1}.ex.pc', \ 'co_{co_{m-1}.p}.ex.pc$
7 $Bg \leftarrow 0$
8 $Fst \leftarrow 1$
9 $cntPNET \leftarrow 0$
10 $cntBET \leftarrow 0$
11 **else**
12 **if** $cntBET = 0$ **then**
13 $co_m.en_0.pc \leftarrow co_{co_m.p}.en_0$
14 **else**
15 $co_m.en_0.pc \leftarrow co_{m-1-cntPNET}.ex$
16 **end if**
17 **if** $cntPNET > 0$ **then**
18 $co_m.ex.pc \leftarrow co_{m-1}.ex$
19 **end if**
20 **end if**
21 **if** Fst **then**
22 $co_m.a_0 \leftarrow co_m.en_0.pn$" *occurs every* $[$"$F * cb_m.ap$", "$F * cb_m.ap$"$]$ "$cb_m.tu$" *with offset* $[$"$F * cb_m.op + cb_m.of$", "$F * cb_m.op + cb_m.of$"$]$ "$cb_m.tu$"."
23 $co_m.g_0 \leftarrow$ "*Reaction* ("$co_m.en_0.pn$", "$co_m.ex.pn$") *within* $[0,$ "$cb_m.bg$"$]$ "$cb_m.tu$"."
24 **else**
25 $co_m.en_1.pn \leftarrow PEntry$
26 $co_m.en_1.pc \leftarrow co_{co_m.p}.en_0$
27 $co_m.a_0 \leftarrow co_m.en_0.pn$" *occurs every* $[$"$F * cb_m.ap - Bg$", "$F * cb_m.ap + Bg$"$]$ "$cb_m.tu$" *with offset* $[$"$F * cb_m.op + cb_m.of - Bg$", "$F * cb_m.op + cb_m.of$"$]$ "$cb_m.tu$"."
28 $co_m.a_1 \leftarrow co_m.en_1.pn$" *occurs every* $[$"F", "F"$]$ "$cb_m.tu$" *with offset* $[$"$cb_{cb_m.p}.of$", "$cb_{cb_m.p}.of$"$]$ "$cb_{cb_m.p}.tu$"."
29 $co_m.g_0 \leftarrow$ "*Reaction* ("$co_m.en_0.pn$", "$co_m.ex.pn$") *within* $[0,$ "$cb_m.bg + Bg$"$]$ "$cb_m.tu$"."
30 **if** $co_{co_m.p}.nm.tp =$ PET **then**
31 $co_m.g_1 \leftarrow$ "*Reaction* ("$co_m.en_1.pn$", "$co_m.ex.pn$") *within* $[$"$cb_m.of - Bg$", "$cb_m.of + cb_m.bg$"$]$ "$cb_m.tu$"."
32 **else**

```
33              co_m.g_1 ← "Reaction ("co_m.en_1.pn", "co_m.ex.pn") within [0, "cb_m.bg + Bg
                    "] "cb_m.tu"."
34          end if
35      end if
36      if cntPNET = 0 then
37          Bg ← Bg + cb_m.bg
38      end if
39      m ← m + 1
40      cntPNET ← cntPNET + 1
41  end procedure
```

Declaration

I hereby declare that this thesis titled, "Legacy Software Migration based on Timing Contract aware Real-Time Execution Environments" was composed by myself, and that the work contained herein is my own except where explicitly stated otherwise in the text. This thesis has also not been submitted in whole or part for any other degree or professional qualification.

Oldenburg, 2020-01-26

Irune Yarza Perez