# Static/Dynamic Real-Time Legacy Software Migration – A Comparative Analysis

Irune Yarza
Mikel Azkarate-askatsua
Peio Onaindia
iyarza@ikerlan.es
mazkarate@ikerlan.es
ponaindia@ikerlan.es
Ikerlan Technology Research Centre,
Dependable Embedded Systems Area
Arrasate-Mondragón, Spain

Philipp Ittershagen
Kim Grüttner
philipp.ittershagen@offis.de
kim.gruettner@offis.de
OFFIS - Institute for Information Technology
Oldenburg, Germany

Wolfgang Nebel
nebel@informatik.uni-oldenburg.de
C.v.O. Universität Oldenburg
Oldenburg, Germany

## ABSTRACT

Evolution to next generation embedded systems is shortening the obsolescence period of the underlying hardware. As this happens, software designed for those platforms (a.k.a., legacy code), that might be functionally correct and validated code, may be lost in the architecture and peripheral change. As embedded systems often have Real-Time (RT) computing constraints, the legacy code retargeting issue directly affects RT systems. Binary translation techniques have been widely applied for legacy code migration. However, there are just a few works that consider RT legacy code. Therefore, this paper presents a static and a dynamic binary migration approach (based on QEMU and Rev.ng respectively) and analyzes and compares their suitability as RT code migration solutions. The comparison shows that among the proposed solutions, the static is the most appropriate for short-running RT legacy code, since it ensures lower translation overhead and a more deterministic timing behavior. Instead, the dynamic approach might be a suitable solution for RT legacy code with long periods (over 0.01 s) and mainly composed of complex floating point computations, since the dynamic translation and optimization overhead is not that significant on long-running benchmarks and the static translation implies a great slowdown on benchmarks containing floating point operations.

## CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

## KEYWORDS

binary translation, legacy code, real-time systems, retargeting

**ACM Reference Format:**
Irune Yarza, Mikel Azkarate-askatsua, Peio Onaindia, Philipp Ittershagen, Kim Grüttner, and Wolfgang Nebel. 2020. Static/Dynamic Real-Time Legacy Software Migration – A Comparative Analysis. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '20), January 21, 2020, Bologna, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3375246.3375257


*RAPIDO '20, January 21, 2020, Bologna, Italy*

ACM ISBN 978-1-4503-7777-5/20/01…$15.00
https://doi.org/10.1145/3375246.3375257

## 1 INTRODUCTION

Companies within the embedded systems 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 there is a need to deal with legacy code. Legacy code is characterized by some particular properties: it usually runs on obsolete Hardware (HW) which is slow and expensive to maintain [13], uses customized and deprecated toolchain(s), has no or outdated documentation [11], and it is essential for the company [2] since it comprises business knowledge [12].

Due to the fact that classical process models focus on the development stage of software life-cycle instead of operation-maintenance stages, the process of updating legacy systems is usually complex, error-prone, time-consuming and requires high cost investment. In response to this problem, research efforts have provided several solutions. Nonetheless, when it comes to legacy software migration, Binary Translation (BT) appears to be a standard approach, as the binary that runs on the legacy HW can be ported to a new HW platform without a considerable expense of time, effort and money.

Although BT has been successfully applied for legacy software migration, it is necessary to consider that when dealing with RT legacy code migration, not just the functional properties, but also the timing behavior has to be preserved. To the authors knowledge, Cogswell [4] and Heinz [7] are the only ones who considered timing on their proposed retargeting solutions. However, they have limitations regarding their portability. Therefore, industry still needs a low-overhead embedded RT software retargeting solution that can be easily ported to different source and target architectures.

In the direction to solve this problem, the overall goal of this research line is to enhance the latest low-overhead machine-adaptable static/dynamic BT tools with the ability to preserve the timing behavior on the translated binary. This will enable the migration of RT embedded legacy code to a new HW platform with guaranteed

RT performance. To this end, two approaches were considered: (1) a dynamic approach based on Quick EMUlator (QEMU) [1] and (2) a static approach based on Rev.ng [5].

As a first step on the research, in a previous publication [15] we analyzed the suitability of QEMU, for its use in a RT property conserving retargeting process. In the same vein, this article analyzes and compares the suitability of the static and dynamic approaches (on a chosen test/evaluation set-up) for their use in a RT property conserving retargeting process. Therefore, the main contribution of this paper is the construction of a test environment to check whether is better to choose a static or a dynamic approach to port a particular RT legacy binary to a new architecture being able to reproduce the timing behavior on the legacy Instruction Set Architecture (ISA). The detailed technical contributions of this paper are:

- A survey on existing static and dynamic code translation techniques heeding portability, embedded systems or RT legacy code.
- A description of the static and dynamic RT legacy code migration approaches.
- A feasibility study and comparative analysis of the described static and dynamic RT legacy code migration solutions.

The remainder of this paper is organized as follows. An overview of related work in the area of machine-adaptable static and dynamic code translation techniques for embedded systems is provided in Section 2. Then, Section 3 presents the proposed solution and describes the static and dynamic approaches considered on this research for RT legacy code migration. In order to perform a feasibility study of the proposed approaches, Section 4 describes the construction of a test framework with means for high-resolution execution time measurement of periodically triggered software. The obtained experimental results are thus evaluated on Section 5. Finally, Section 6 gives an outlook on future work and a conclusion.

## 2 RELATED WORK

BT techniques have been widely studied and developed in the last decades. Table 1 provides a summary of the related work reviewed in [15]. Whereas this section analyzes the related work with a focus on aspects such as portability, and RT legacy code and system-level code support. Reviewed work and this section

**Table 1: Related work summary. Cross-platform BT tools are analyzed according the following four aspects: static/-dynamic translation, portability, and RT legacy code and system-level code support.**

| Name | Static/Dynamic | Machine-adaptable | RT legacy Code | User-/System-level |
|---|---|---|---|---|
| **TIBBIT** [4] | Static | - | ✓ | System-level |
| **Heinz** [7] | Static | - | ✓ | System-level |
| **UQBT** [3] | Static | Source & Target | - | User-level |
| **UQDBT** [10] | Dynamic | Source & Target | - | User-level |
| **QEMU** [1] | Dynamic | Source & Target | - | User- & System-level |
| **DisIRer** [8] | Static | Target | - | User-level |
| **CrossBit** [14] | Dynamic | Source & Target | - | User-level |
| **Rev.ng** [5] | Static | Source & Target | - | User-level |
| **LLBT** [9] | Static | Target | - | User-level |

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. So given that BT tools are highly dependent on the source and target architectures, researchers adopted the general approach of portable compilers, where machine-dependent and machine-independent concerns are separated, to provide a **machine-adaptable** binary translator. The first machine-adaptable solution, UQBT [3], supports multiple source and target machines by using specifications that describe the ISA and Operating System (OS). Unlike UQBT, most machine-adaptable solutions provide multiple front-ends and benefit from a retargetable compiler to provide support for multiple target ISA.

When dealing with time sensitive code migration, not just its functional behavior, but also the timing behavior of time critical tasks has to be preserved. From the related work, just [4] and [7] 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 behavior. 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 Worst Case Execution Time (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-Dygital Converter (ADC); serial, Ethernet or CAN controller; sensor/actuator), also known as **system-level code**. However, most of the approaches on the State of The Art (SOTA) propose migration solutions for **user-level code**, where the underlying OS's Application Programming Interface (API) abstracts the low-level code from the application. The approaches presented in [4] and [7] support system-level binaries, whereas, QEMU that was first designed for Linux machine emulation, now supports also system-level code.

## 3 RT LEGACY CODE MIGRATION

In order to provide a migration path to RT legacy software, two approaches have been considered: a dynamic approach based on QEMU [1] machine emulator and a static approach based on Rev.ng [5] binary instrumentation framework. The following subsections describe the characteristics of legacy system to be ported and present the assumptions and current constraints of the proposed migration solution. The proposed static and dynamic migration approaches are then described in detail, setting the focus on how each of the tools performs the BT process and how each of them overcomes the main portability aspects.

### 3.1 Legacy System

The left side of Figure 1 depicts the legacy system architecture, which follows the typical pattern of a reactive control system. The application is periodically triggered (at $t_0$, $t_1$, ...) to read new data from the sensors and update the actuators after a period of time

($dt$). For an appropriate behavior of the system under control, the duration of the application ($dt$) must be below the execution period ($t_{n+1} - t_n$). Moreover, every action on the legacy application that implies information exchange (e.g. read/write from/to Input/Output (I/O) device buffer or shared variables) is also likely to be timing critical.

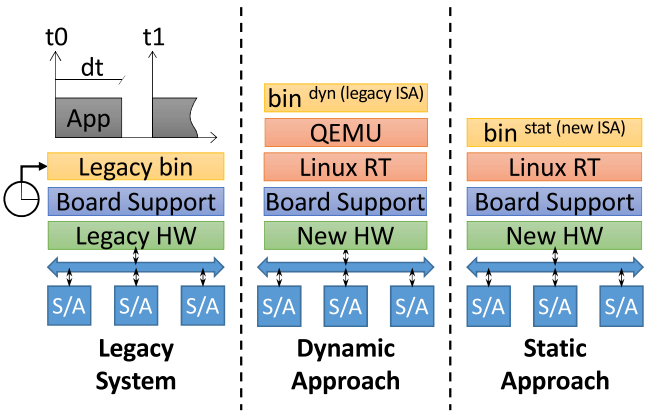


**Figure 1: Runtime architecture of RT legacy code running on the legacy system (left hand), dynamic approach (middle), and static approach (right hand).**

## 3.2 Assumptions and Constraints

The legacy code block that needs to be ported, is treated as a gray box that is being reused with little knowledge of its implementation. The source code is available, however, there is a preference to keep it unmodified due to possible unfamiliarity with the code and to increase the usability of the approach.

The current proposal only considers applications that have manually adapted I/O accesses to the new HW platform and do not make use of any HW timer (i.e. we currently only consider pure computational applications and do not consider I/O virtualization between the legacy and new HW platform). Moreover, the current approaches provide means to implement the periodic execution loop, but do not yet support time enforcement at a finer granularity. These constraints will be lifted in future work.

## 3.3 Dynamic Approach

The dynamic approach takes advantage of QEMU [1] to translate legacy code on run-time. As I/O and timer virtualization is not supported, in order to access the host timer and implement a periodic execution loop (without re-launching QEMU), the legacy application and QEMU's source code have to be adapted before translation. The adapted legacy application is then compiled for the legacy processor and runs on top of adapted QEMU. The translator, is launched on top of a minimal Linux distribution, which has been configured using the PREEMPT_RT patch [1]. The adapted QEMU is launched with the highest allowed priority [2]. In the center of Figure 1 the described dynamic approach run-time architecture is shown.

The following subsections describe how the legacy application is adapted, which is QEMU's translation process and how QEMU's source code is adapted to reach our goal.

*3.3.1 Adapt legacy application.* In the legacy application, I/O accesses are replaced with I/O variables and a particular approach is followed to periodically launch the application. An application container is defined, which initializes state variables (as it is done in the legacy application) and sets an infinite execution loop. Inside this loop, first an empty function call is inserted[3], `start_period()`, that will allow the identification of the period start point from QEMUś translation process. Then, input variables are updated (from csv file), the legacy application's behavioral part is executed in a run-to completion manner and the content of output variables is written back (to csv file). Finally, an empty function call is inserted, `end_period()`, to identify the end of the period from QEMU's translation process. Using empty function calls to annotate the legacy application we ensure that there is a branch in the code, consequently this instructions will be the first ones in their corresponding Translation Block (TB).

*3.3.2 QEMU.* The core element in QEMU is its code generator, Tiny Code Generator (TCG), which is responsible for the dynamic translation of target source code into host machine code. As a machine-adaptable Dynamic Binary Translation (DBT), TCG adopts the general approach in portable compilers. Therefore, the source code TBs are first translated into tiny code instructions, a machine independent Intermediate Representation (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 2 illustrates in a flow diagram QEMU's run-time behavior. 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. Afterward, `tb_find()` function searches the next TB according to the current Program Counter (PC) value. If no TB is found, target machine code is generated through `tb_gen_code()` function, which subsequently call functions `gen_intermediate_code()` to translate source code into tiny

[1] The main purpose of PREEMPT_RT patch is to improve the RT behavior 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.

[2] The highest allowed priority is 98, since PREEMPT_RT uses 99 as the priority for the kernel task sets and interrupt handler.

[3] An empty function call is a call to a function whose body is empty.

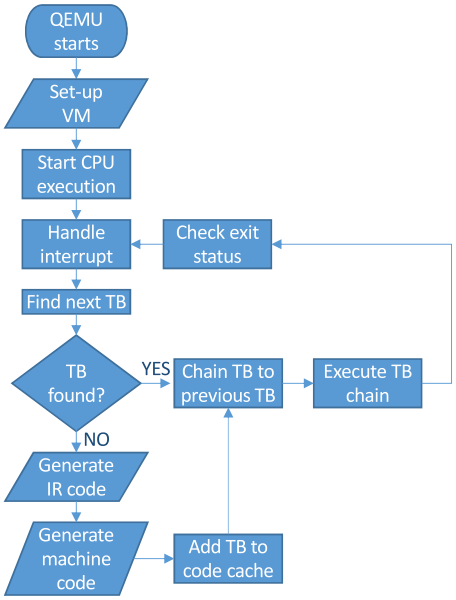


**Figure 2: QEMU's DBT flow diagram.**

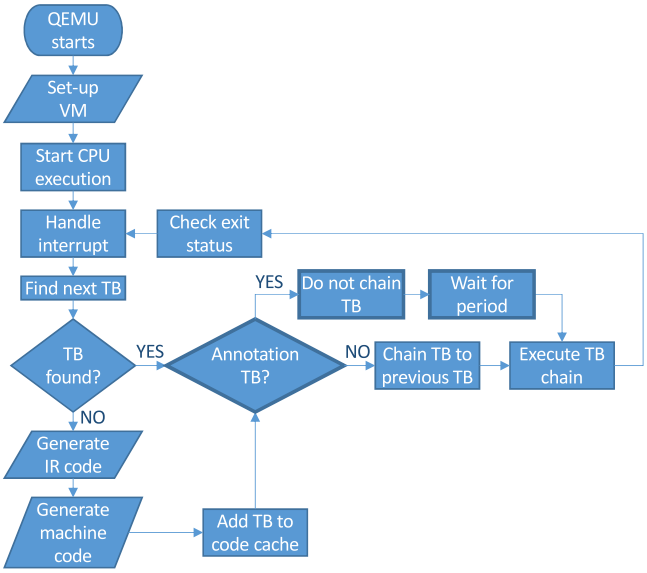


**Figure 3: QEMU's DBT flow diagram adapted to establish the periodic execution loop.**

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.

*3.3.3 Adapt QEMU.* In order to establish the periodic execution loop, apart from the legacy application, QEMU's source code also has to be adapted in such a way that it identifies the annotations (empty function calls) in the legacy application and implements the periodic loop. Figure 3 illustrates in a flow diagram the run-time behavior of the adapted QEMU. QEMU identifies TBs with the PC value of the first instruction in the block. Since we annotated the legacy application with empty function calls, we ensured that these instructions will be the first instruction in the TB. So, if the generated/found TB corresponds to the annotation PC value (start_period_pc() or end_period_pc()) an auxiliary code is inserted that, based on Linux high resolution timers, gets the start/end time (saved in pStart_dyn or pEnd_dyn), measures the duration, compares it with the period and waits until they are equal. 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 these TBs are never chained to the previous one.

## 3.4 Static Approach

The static legacy code migration approach employs Rev.ng [5], to translate a statically linked Linux binary into equivalent target machine code. As I/O and timer virtualization is not supported, the legacy application has to be adapted. First of all, accesses to I/Os are replaced with I/O variables. Then, an application container is defined, which initializes the state variables (as it is done in the legacy application) and creates a periodic execution loop that contains the adapted legacy application's behavioral part which is executed in run-to completion mode. The application container which includes the legacy code is compiled using a Linux toolchain for the legacy architecture. The statically generated binary is then translated off-line, before run-time. Once translated, the new binary runs with priority 98 (highest allowed) on top of a minimal Linux distribution that has been configured with the PREEMPT_RT patch. On the right hand of Figure 1 the run-time architecture of the static migration approach, as it has been described, is shown.

*3.4.1 Rev.ng.* Rev.ng is a binary analysis framework whose core element is, Revamb Static Binary Translation (SBT) tool, which combines the benefits of QEMU, with those of Low Level Virtual Machine (LLVM). LLVM is a compilation framework that provides source and target independent optimization 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.

Rev.ng 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 [4] and LLVM [5] 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 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 4 depicts the translation process of Rev.ng tool suite, which combines the use of QEMU's front-end and LLVM.

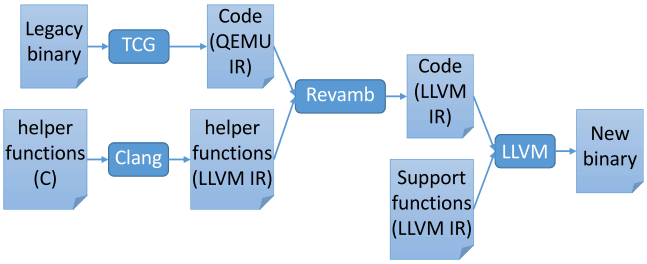


**Figure 4: Rev.ng's SBT process combining the use of QEMU, Revamb and LLVM.**

## 4 FEASIBILITY STUDY

The feasibility study assess the static and dynamic migration approaches described above with respect to timing. To do so, a test framework has been constructed, which provides means to measure the execution time of a selection of WCET representative benchmark programs, provided by the Mälardalen WCET research group [6] running on the legacy platform and on the new HW platform using both migration approaches, static and dynamic. The obtained results are then analyzed and compared in Section 5.

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 an 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 measure execution time on the ARM processor we used Xilinx's Board Support Package (BSP) to access the global timer counter, whereas to perform the measurements on the Intel Atom processor the Linux high-resolution timer has been used.

### 4.1 Dynamic Instrumentation

For the timing assessment of the dynamic approach, benchmarks need to be instrumented. However, the dynamic approach instrumentation solution is a twofold technique. On the one hand, the source code is annotated with empty function calls (`start_time()` and `end_time()`) to ease the start/end detection in QEMU. Using an empty function call we ensure that there is a branch in the code, consequently this instruction will be the first in the TB and we will be able to detect it though the PC. On the other hand, QEMU source code has been modified to integrate start/end PC detection (`start_time_pc` and `end_time_pc`) and perform the execution time measurements. When launching QEMU, the `start_time_pc` and `end_time_pc` values corresponding to the running benchmark are passed through arguments. The function in charge of finding the next TB, `tb_find`, identifies `start_time_pc` and `end_time_pc` and computes the duration. Moreover, as previously mentioned, QEMU chains consequently executed TBs to avoid context switch cost. As a consequence, start and end TBs would be chained to former TBs and control would not return to the execution manager. Therefore, `tb_find` function has been altered to avoid start and end TB chaining.

## 5 FEASIBILITY RESULT ANALYSIS

The feasibility survey compares the execution time of the Mälardalen WCET benchmarks [6] running on top of the legacy HW platform and the new HW using both, dynamic and static migration approaches. Given that the selected benchmarks contain a great variety of algorithms (including loops, nested loops, use of array and/or matrices and use of floating point operations), we get a wide analysis of the timing behavior of the proposed static and dynamic solutions.

### 5.1 Platform configuration

For the execution time analysis on the legacy HW (ZC702), the Vivado Zynq example project is used. The generated bitstream is exported to Software Development Kit (SDK), where benchmarks are compiled (without any optimization [7], -O0).

The same example project is used to run the legacy code on the new HW platform (MinnowBoard) through the dynamic approach. However, as explained before, due to QEMU's start-up procedure, the code has to be compiled and linked to be placed at the OS starting memory location, which in the case of armv7 architecture is 0X10000. As well as for the legacy HW, the benchmarks have been compiled without optimization.

[4] 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.

[5] 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.

[6] Despite the fact that the Cortex-A9 processor is not a legacy HW 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.

[7] This is a common practice in RT systems where the WCET is important for reliability or correct functional behavior

Regarding the static legacy code migration approach, Rev.ng provides a cross-toolchain for each of the supported input architectures. Therefore, the statically linked Linux input binary has been generated using the corresponding toolchain (armv7a-hardfloat-linux-ublibceabi-gcc) and without applying any optimization. Then, the input binary has been statically translated using the *translate* script provided with Rev.ng tool suite.

## 5.2 Evaluation process and results

To get the results, each benchmark has been executed 15000 times (statistically relevant enough) on the legacy and new HW platforms (using dynamic and static migration solutions) while collecting timing data. Given that we are targeting the migration of a reactive control system where the application is periodically triggered, QEMU is launched only once executing the benchmarks periodically and the first benchmark runs are 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[8]. Together with the WCET benchmarks an empty application[9] has also been analyzed. Measuring the empty application execution time provides means 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.

*5.2.1 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 5 shows the collected data distribution with a zoom in the maximum execution time result area. Table 2 contains the minimum, maximum, average, standard deviation and 99%-quantile of the collected data.

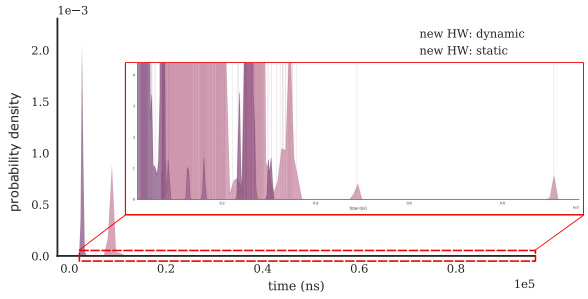


**Figure 5: Distribution of execution time data collected running the empty application on the new HW platform following a dynamic/static translation process.**

Results show that, as expected, the average translation overhead of the DBT 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.

**Table 2: Maximum, minimum, average, standard deviation and 99%-quantile of the measured execution time when running multiple times the empty application on the new HW platform following a dynamic/static translation process.**

| | 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 |

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 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 5, are more 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.

*5.2.2 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 HW 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 6. 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.

When analyzing the results, benchmarks are classified into short- and long-running according to their average execution time on the legacy HW. 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 3). Based

[8] An analysis of the (quantified) performance during this warm-up is out of the scope of this work.
[9] We consider an empty application that whose main function does not contain any instruction.

on the information provided by the Mälardalen WCET research group [6], we have classify benchmarks depending on the type of operations they contain: (1) complex computations, (2) simple computations or (3) control flow statements. 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 behavior.

**Table 3: 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 = composed of complex computations. SC = composed of simple computations CF = composed of control flow statements.**

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

Results show that most of the benchmarks that have been analyzed 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 optimization 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 analyzing the average execution time ratio between dynamically translated binaries running on the new HW and legacy binaries running on the legacy HW, 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 ratio between statically translated binaries running on the new HW 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.

## 6 CONCLUSIONS AND FUTURE WORK

This work aimed at describing the proposed static and dynamic embedded RT legacy code migration approaches and at performing a feasibility analysis of both solutions. The dynamic approach is based on QEMU machine emulator, whereas the static approach is based on Rev.ng binary instrumentation framework. When migrating a RT application, not just its functional properties but also the timing behavior needs to be preserved. Therefore, the feasibility study compares both migration approaches with regards to measured execution time. To this end, a test framework has been constructed, which provides means to measure the execution time of code ported using both migration approaches. The test environment has been implemented on top of an Intel Atom E3826 processor, where a selection of WCET representative benchmarks compiled for ARM Cortex-A9 have been ported. 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 RT legacy code. Whereas the dynamic approach might be a better choice when porting RT legacy code with long periods (over 0,01s).

As already mentioned in the introduction, this work described early results. Future work will provide means to preserve the timing behavior of the legacy code on the new HW platform. To this end, it is necessary to define the timing constraints that the system has to meet and their granularity and the execution time control mechanism that will be integrated in the new ISA. The timing enforcement solution together with I/O virtualization implemented on an appropriate BT system (dynamic or static, depending on the characteristics of the legacy software to be ported) will provide means to enforce the legacy timing requirements on the new HW platform and a way of time sensitive interaction between the control system and the external environment.

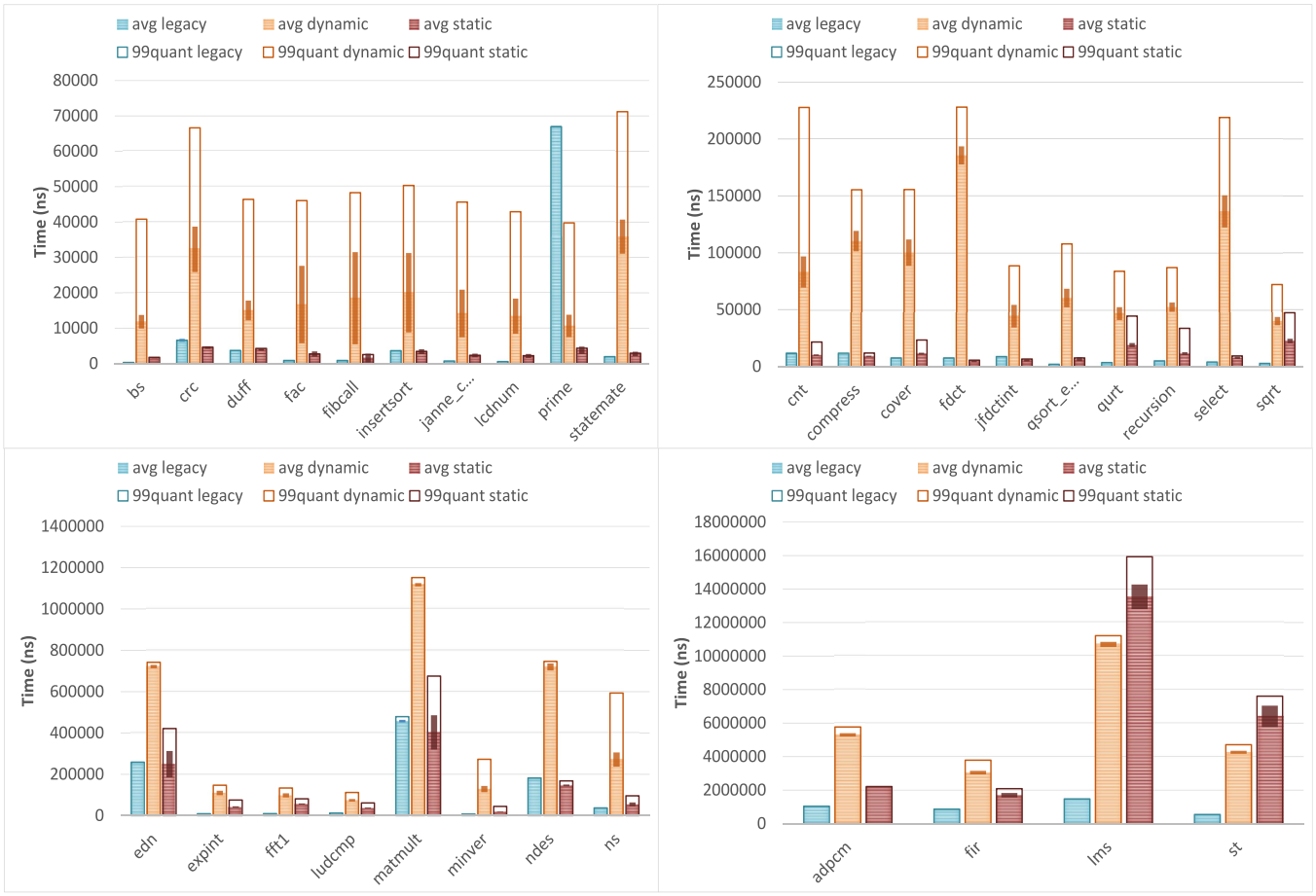


**Figure 6: Timing results of benchmarks running on the legacy and new HW platforms: static vs. dynamic translation.**

## 7 ACKNOWLEDGMENTS

The authors would like to thank the Rev.ng tool suit developers for supporting them with the Rev.ng tool and providing them access to their private repository.

## REFERENCES

[1] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track. 41–46.

[2] Keith Bennett. 1995. Legacy systems: Coping with success. IEEE software 12, 1 (1995), 19–23.

[3] C. Cifuentes and M. Van Emmerik. 2000. UQBT: adaptable binary translation at low cost. Computer 33, 3 (2000), 60–66. https://doi.org/10.1109/2.825697

[4] Bryce Cogswell and Zary Segall. 1995. Timing insensitive binary to binary translation of real time systems. In Workshop on Architectures for Real-Time Applications, ISCA.

[5] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In CC 2017. ACM, 3033028, 131–141. https://doi.org/10.1145/3033019.3033028

[6] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In WCET2010, Björn Lisper (Ed.). OCG, Brussels, Belgium, 137–147.

[7] Thomas Heinz. 2008. 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, 1–4.

[8] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. 2010. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. ACM Transactions on Architecture and Code Optimization (TACO) 7, 4 (2010), 18.

[9] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems. ACM, 51–60.

[10] David Ung and Cristina Cifuentes. 2000. Machine-adaptable dynamic binary translation. In ACM SIGPLAN Notices, Vol. 35. ACM, 41–51.

[11] Christian Wagner and Christian Wagner. 2014. Model-Driven Software Migration. Springer.

[12] M. Wahler, R. Eidenbenz, C. Franke, and Y. A. Pignolet. 2015. Migrating legacy control software to multi-core hardware. In Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. 458–466. https://doi.org/10.1109/ICSM.2015.7332497

[13] Bing Wu, Deirdre Lawless, Jesus Bisbal, Jane Grimson, Vincent Wade, Donie O'Sullivan, and Ray Richardson. 1997. Legacy system migration: A legacy data migration engine. In Proceedings of the 17th International Database Conference (DATASEM'97). 129–138.

[14] Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, and Bo Liu. 2010. Crossbit: a multi-sources and multi-targets DBT.

[15] Irune Yarza, Mikel Azkarate-askasua, Kim Grüttner, and Wolfgang Nebel. 2018. Real-Time Capable Retargeting of Xilinx MicroBlaze Binaries using QEMU: A Feasibility Study. In Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools. ACM, 3180671, 1–8. https://doi.org/10.1145/3180665.3180671