RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications

Mattia Nicolella
Boston University, USA
mnico@bu.edu

Shahin Roozkhosh
Boston University, USA
shahin@bu.edu

Denis Hoornaert
TU München, Germany
denis.hoornaert@tum.de

Andrea Bastoni
TU München, Germany
andrea.bastoni@tum.de

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

ABSTRACT

Benchmarking is crucial for testing and validating any system, including—and perhaps especially—real-time systems. Typical real-time applications adhere to well-understood abstractions: they exhibit a periodic behavior, operate on a well-defined working set, and strive for stable response time, avoiding non-predicable factors such as page faults. Unfortunately, available benchmark suites fail to reflect key characteristics of real-time applications. Practitioners and researchers must resort to either benchmark heavily approximated real-time environments or re-engineer available benchmarks to add—if possible—the sought-after features. Additionally, the measuring and logging capabilities provided by most benchmark suites are not tailored "out-of-the-box" to real-time environments, and changing basic parameters such as the scheduling policy often becomes a tiring and error-prone exercise.

In this paper, we present RT-bench, an open-source framework adding standard real-time features to virtually *any* existing benchmark. Furthermore, RT-bench provides an easy-to-use, unified command-line interface to customize key aspects of the real-time execution of a set of benchmarks. Our framework is guided by four main criteria: 1) cohesive interface, 2) support for periodic application behavior and deadline semantics, 3) controllable memory footprint, and 4) extensibility and portability. We have integrated within the framework applications from the widely used SD-VBS and IsolBench suites. We showcase a set of use-cases that are representative of typical real-time system evaluation scenarios, and that can be easily conducted via RT-Bench.

CCS CONCEPTS

• **Computer systems organization** → **Real-time systems**; • **General and reference** → *Measurement*; *Performance*.

KEYWORDS

framework,interference,open-source,extensible,portable,benchmark suite,real-time,profiling,periodic

RTNS '22, June 7–8, 2022, Paris, France

ACM ISBN 978-1-4503-9650-9/22/06…$15.00
https://doi.org/10.1145/3534879.3534888

ACM Reference Format:
Mattia Nicolella, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato Mancuso. 2022. RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS '22), June 7–8, 2022, Paris, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3534879.3534888

1 INTRODUCTION

In light of the current ever-growing dependence on automated control systems and their increasing complexity, with numerous components consolidated on a chip, safety and determinism certification challenges grow exponentially. At the early stages of development, simulation tools are essential to inspect new designs. However, simulation of the entire system works under the assumption that the employed models accurately represent the actual behavior of the hardware. Unfortunately, complex modern hardware often deviates from textbook models in unpredictable ways as they only reveal partial information on the actual state of the system [35]. Real-time researchers are therefore forced to analyze the system's deployment on the final physical platforms. Benchmark suites are, therefore, frequently used to bridge the gap between simulated and real behavior and empirically assess the ability to deliver real-time performance.

In response to the challenges outlined above, the real-time community has adapted to use miscellaneous sets of techniques to test various system components both before and after integration. The latter is often more demanding as it requires to reason about the interplay between concurrent software components. To that end, numerous benchmarks have been used to test a multitude of characteristics. Unfortunately, however, the lack of standardized benchmarks and general consensus in the testing environment has led to severe fragmentation in testing methodology and poor comparability of results. To make things worse, specific base-platform dependencies and ordeals in porting benchmarks to run on specific hardware are often overlooked challenges that can adversely affect the ability to adopt a given set of benchmarks.

When looking at a real-time platform, metrics such as responsiveness, predictability, and the impact of parallelism are of immediate interest. It is also customary to analyze real-time systems under *typical* load as well as under *stress*. Therefore, a good practice is to use a mixture of both synthetic and realistic benchmarks to construct an informed assessment of how the system reacts to different workload configurations. Despite the abundance of benchmarks of both

types, there is a lack of out-of-the-box applicability to real-time systems.

This paper aims to propose a standard framework, namely RT-Bench, that offers several features designed to meet the needs of researchers and practitioners who are interested in studying the real-time behavior of their systems. Beyond providing an initial set of well-known and publicly available benchmarks[1] that have been adapted to RT-Bench, our goal is to allow the integration of additional applications with as few adaptations as possible in a modular and extensible fashion that emphasizes the importance of good documentation and code reuse. We identify three main categories of functional features: (1) unified launch, control, and reporting interface; (2) adherence to real-time abstractions; (3) cross-platform portability; and (4) automated analysis use-cases.

Unified Interface: RT-Bench provides system designers with the direct ability to control key parameters of benchmark deployment, such as workload composition, scheduling policy and priorities, pinning of applications to CPUs, and enforcement of memory allocation limits, to name a few. At the same time, the framework provides a uniform performance reporting infrastructure that already includes real-time oriented metrics such as job arrival time, deadline, response time, and usage of system resources via performance counters. Lastly, RT-Bench includes a set of automated analysis use-cases aimed at producing key complex metrics in the platform of reference, such as observed WCET, the impact of contention-induced temporal interference, and input-dependent working set size (WSS).

Real-time Abstractions: RT-Bench proposes an infrastructure to transform any monolithic benchmark into a recurrent one with the goal of adhering to the real-time periodic task model. In doing so, it also offers a generic methodology to factor-out typical initialization and tear-down overheads in the acquired measurements. At the same time, real-time applications are often assumed to be deadline-constrained. For this purpose, RT-Bench implements deadline detection and enforcement semantics via job skipping. Moreover, real-time applications are assumed to have well-behaved memory allocation patterns and statically known WSS. RT-Bench offers a generic technique to enforce such semantics even when the original benchmarks make use of dynamic memory allocation (e.g., via `malloc` and `free`) and without requiring code refactoring.

Cross-platform Portability: RT-Bench only uses APIs from the POSIX standard to allow deployment on an extensive range of Operating Systems (OS) and bare-metal software stacks (e.g. Newlib [39]). We decouple it from any system-specific limitations through user application-level implementation. The only exception is user-space interaction with platform-specific cycle counters for which support is provided in all the leading architectures.

We test our framework by adapting to the state-of-the-art benchmark suits as we explore their characteristics in the remainder of this paper. A proof-of-concept integration of IsolBench [37] and San Diego Vision Benchmark Suite (SD-VBS) [38] into the proposed framework is provided. The open-source RT-Bench implementation is available at https://gitlab.com/bastoni/rt-bench.

[1] San-Diego Vision Benchmark suite [38]

2 RELATED WORK

With the ever-growing explosion in the complexity of embedded computing platforms, performance characterization and prediction have become increasingly more challenging. Moreover, to reach a conclusive assessment regarding the temporal properties of a system, it is crucial to test the system's behavior under different workloads. The real-time community has adopted a number of strategies to obtain indicators of the system behavior through benchmarking.

This section provides a comparative survey of popular benchmark suites used in the real-time community. The survey is summarized in Table 1. These suites can be categorized into three groups:

- **Synthetic Benchmarks (SB)** that will stress a particular element or aspect of the system under analysis.
- **Pragmatic Benchmarks (PB)** batch processing tasks mimicking realistic workload such as image processing, signal processing, physics simulation, and matrix multiplication.
- **Full-Scale (FS)** real-time applications containing a mixture of hard/soft/non-realtime jobs with both periodic and non-periodic tasks to be executed on embedded systems for full-system concrete timing verification.

Popular **Synthetic** benchmarks include IsolBench [37] (used in [14, 15, 23]), the RT-Test [3] suite and the RTEval [4] benchmark. IsolBench is a collection of memory workloads used to analyze the memory bandwidth and latency. It supports periodic execution, but it does not have a comprehensive interface logging data on each period. The RT-Test suite is a set of benchmarks to profile the responsiveness of the Linux kernel. The RTEeval benchmark relies on RT-Test to measure the performance of the Linux kernel under specific workloads.

The most accustomed **Pragmatic** benchmark suites include: TACLeBench [13] (used in [25, 27, 36]), San Diego Vision Benchmarks [38] (used in [6, 14, 32]), Mälardalen [17] (used in [8, 28, 29]), several versions of SPLASH [5, 33, 40] (used in [16, 26, 41]), EEMBC [11] (used in [20, 21]), and MiBench [18] (used in [12, 24]). TACLeBench has been designed with portability in mind for most of the benchmarks, as they target WCET analysis. This suite regroups other synthetic benchmarks such as Papabench [30]. Hence, TACLeBench lacks a homogeneous interface. SD-VBS performs general image processing and vision-related jobs, and it aims at offering maximum portability. However, to be used as embedded applications, the benchmarks must be adapted for periodic execution. Dynamic memory allocation is also widely used, which further hinders their temporal determinism. The Mälardalen benchmarks share many components with TACLeBench. However, they also suffer from some of the same shortcomings. The Mälardalen benchmarks are not designed for periodic execution. Instead, they are mainly designed to be good targets for static WCET analysis. EEBMC is a selection of benchmarks specifically targeting embedded devices of different types, ranging from mobile to automotive systems. MiBench is similar to EEMBC, but was created to address its shortcomings. Finally, the SPLASH benchmark suite is a collection of benchmarks tailored to parallel execution and WSS analysis from Pragmatic models. On a similar flavor, suites like the PARSEC [9] (used in [16]) and Rodinia [10] suites (used in [7, 31]) represent an interesting alternative as they specifically target parallel execution,

Table 1: Benchmark suites comparison (ALL = Synthetic, Pragmatic, and Full-Scale)

Benchmark suite	Type	Periodic exec.	Cross Platform	Unified Interface	Dead. status	Exec. Time	Utili-zation	Density	Profiled WCET	Profiled WSS	Perf. counters	Memory Profiler
TACLeBench [13]	ALL	-	-	✗	✗	✗	✗	✗	✗	✗	✗	✗
SD-VBS [38]	PB	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Mälardalen [17]	PB	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
SPLASH [5, 33]	PB	✗	CUDA	✗	✗	✓	✗	✗	✗	✓	✗	✗
Rodinia [10]	PB	✗	✓	✗	✗	-	✗	✗	✗	✗	✗	✗
PARSEC [9]	PB	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
IsolBench [37]	SB	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
EEBMC [11]	PB	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
MiBench [18]	PB	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
PapaBench [30]	FS	✓	AVR	✗	✗	✗	✗	✗	✗	✗	✗	✗
RT-Tests [3]	PB/SB	some	✓	✗	✗	✓	✗	✗	some	✗	✗	✗
RTEval [4]	SB	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗
RT-bench	PB/SB	✓	ARM/x86	✓	✓	✓	✓	✓	script	script	Perf	Aarch64

with Rodinia even offering support for GPUs and heterogeneous systems.

Full-scale real-time applications mostly come from the WATERS industrial challenge [19], formerly called Formal Methods for Timing Verification (FMTV). Full-scale applications are, for the most part, periodic, but extending them is complex, and they might not have broad multi-platform support. An example of a full-scale real-time application is the Papabench [30] benchmark, which encapsulates all the main components of a control system for UAVs.

Table 1 summarizes the essential characteristics (columns) of the surveyed benchmarks (rows). The reported characteristics include (1) the *Type* of benchmark the suite offers according to the aforementioned categories, (2) the capability to be executed in a *Periodic* fashion, (3) the provided cross-platform support, (4) whether it provides a unified interface with other suites, and (5) the metrics natively reported by the suites. For the latter, this includes, from left to right, whether the deadline has been met, the execution time, the processor utilization[2], the density[3], the empirically observed WCET, the ability to collect and report end-to-end hardware events obtained though performance counters (e.g., cache accesses), and the ability to monitor the trend of observed hardware events throughout the execution. Note that categories for which no clear-cut answer exists are marked in orange. This is the case for the platforms supported, and the test provided by RT-Bench noted as *script*, meaning that they rely on high-level tools. The table highlights the necessity of a framework that is specifically designed for the analysis of real-time systems. Indeed, the core philosophy of the proposed RT-Bench framework is to provide an infrastructure to build a reference set of real-time benchmarks with standard functionalities. As a first step in this direction, RT-Bench already offers key analysis tools such as execution-time distribution analysis, WSS examination, and sensitivity to interference. Moreover, with RT-Bench, existing benchmarks can be integrated to execute periodically and to exhibit controlled memory footprint with minimum re-engineering effort.

[2]Computed as measured execution time over the period.
[3]Computed as measured execution time over the relative deadline.

3 DESIGN GOALS AND OVERVIEW

As presented in Section 2, a real-time analysis should ideally be conducted using a large set of benchmarks with different characteristics to provide a comprehensive understanding of the (real-time) performance of the system under analysis. With that respect, the objective of the proposed RT-Bench framework is three-fold.

Common and cohesive interfaces. The use of benchmark suites is widespread in the real-time community, and it is not rare for multiple suites to be jointly used in a given study. These suites are, in most cases, contributions from distinct individuals having particular focuses, ranging from CPU- or memory-bound to CPU- or memory-intensive applications. Unfortunately, while this diversity is a strength, it entails a fragmentation of the parameters available (e.g., assigned processing units, scheduling policy), the metrics reported (e.g., response time, working set size), and the overall user experience. RT-Bench aims at bridging this gap by homogenizing the available features and the reports generated for any benchmark by offering a unified and coherent interface.

Adherence to Real-Time System Abstractions We aim to incorporate, within the proposed RT-Bench framework, a set of features that are in line with the typical models and assumptions used for research, analysis, and testing of real-time systems. We consider this objective of paramount importance and a clear distinctive factor compared to the surveyed suites. RT-Bench is deliberately designed from the ground up to transform any one-shot benchmark into a periodic application with deadline enforcement and job-skipping semantics, with compartmentalized one-time initialization and teardown routines, so to obtain precise measurements. In addition, any benchmark integrated within RT-Bench natively features options to control allocation on a specific set of cores; to be assigned a scheduling policy, and to limit and pre-allocate memory. These features effectively align the behavior of RT-Bench applications with a critical mass of assumptions and abstraction that are customary when analyzing real-time systems.

Extensibility, portability, and usability. We carefully designed the proposed framework, RT-Bench, to be easily extensible, portable, and practical. We deliberately implemented the RT-Bench core features to operate in user space so as to decouple our framework

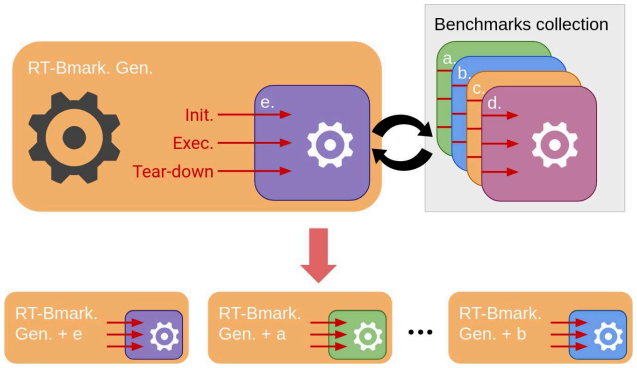


Figure 1: The RT-Benchmark Generator can simply be extended with any benchmark from a collection (e.g., *a*, *b*, *c*, *d*, and *e*) as long as they feature the three harnessing points constituting the interface. It yields RT-Benchmarks version of the adapted third-party benchmarks.

from any system-specific constraints. We do so by leveraging widespread POSIX system-level interfaces. Doing so enables RT-Bench benchmarks to be deployed on a wide range of OS's and bare-metal software stacks (e.g. Newlib). There are only two exceptions to this rule which correspond to two advanced features provided by the framework. The first is the ability to gather timing statistics directly from architecture-specific performance counters. In this case, assembly functions to support x86, Aaarch32, and Aaarch64 systems have already been included. Second, the possibility to also gather statistics from performance counters relies on the Perf [2] infrastructure, which is available by default in typical Linux kernels. To enhance usability, we also provide a complete set of automated build scripts. Likewise, we include a large set of on-the-fly/post-processing scripts.

The RT-Bench framework comprises three specific components: (1) the RT-Benchmark Generator (mandatory), (2) Utils, and (3) Measurements Processing tools. Only RT-Benchmark Generator is mandatory, while the rest are optional. These modules, as mentioned above, are described in the remainder of this section. We further explore their purposes and interaction.

3.1 RT-Benchmark Generator

RT-Bench is designed to be extended with additional third-party benchmarks. Towards this goal, any ported benchmark shall follow the same interface and shall support the same real-time features mentioned in this section.

The conversion to the RT-Bench format is near-transparent, as it only requires the benchmark to be slightly altered to comply with the proposed interface. The interface consists of three functions acting as harness points: (1) *Initialization*, (2) *Execution*, and (3) *Tear-down*. These functions—that must be implemented for a benchmark to be integrated within RT-Bench—are respectively in charge of (1) initializing shared resources such as memory, file descriptors, shared data objects, and the like, (2) executing the main application logic/algorithm, and (3) freeing any of the resources used. Their exact utilization is, from the standpoint of the benchmark, opaquely driven by the RT-Benchmark Generator (see Section 4.1), effectively decoupling real-time features from the design of the application at hand. Further details regarding the implementation are provided in Section 4.2. As illustrated in Figure 1, once the benchmark to be ported is structured following the interface outlined above, the build scripts automate the creation of stand-alone executables that include all the top-level features implemented by the RT-Benchmark Generator. Therefore, encapsulating the desired benchmark within RT-Bench transparently and effortlessly grants it a uniform set of features and a coherent launch interface.

Periodic execution is an essential feature of the framework, as it ensures a periodic execution of the benchmark's main algorithm for a specified amount of iterations—potentially infinitely many. The periodic executions are coherent with the user-specified deadline, meaning if the task does not complete, its successor is not released, and the deadline miss is reported—i.e., RT-Bench applications adhere to the job skipping [34] approach to handle any detected overload conditions.

Core and scheduling policy selection is provided to perform partitioned and clustered multi-core scheduling through pinning to a specific set of cores. The range of execution units and policies available depends on the considered platform. On typical Linux kernels, RT-Bench allows the selection of scheduling policies such as `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, and `SCHED_DEADLINE` and corresponding parameters.

A *deterministic memory layout* is important for real-time applications. Indeed, one often wants to study the memory footprint (or working set size) of the benchmark under analysis and to avoid the overhead of page faults and swapping. While the RT-Bench framework cannot provide a deterministic memory allocation for applications using dynamic memory (e.g., via `malloc` and `free`), it instead enforces a *deterministic memory layout* with two-fold semantics to control memory allocation. When enabled, the user must specify a maximum amount of heap memory to be pre-allocated. All the specified memory is physically allocated (faulted-in) and locked (i.e., made non-swappable) at initialization. Additionally, a watchdog routine is installed to (1) monitor the actual benchmark's footprint at each memory allocation, (2) disable the creation of additional virtual memory regions, and (3) enforce a strict size limit on the heap region, terminating any application exceeding it.

Finally, RT-Bench offers a common *reporting (output) interface* to export the data collected throughout the execution. The metrics listed in Table 2 can be reported in four verbosity levels: (1) error-only; (2) full logging in a CSV file format; (3) full logging on the standard output; and (4) full logging on the standard output in a human-readable format.

Encapsulating the target benchmark within RT-bench means that any ported benchmark benefits from all the aforementioned features. Moreover, they natively display the command-line options to set any of the required parameters. This is ultimately what allows all the applications to share a **standard and coherent launch interface** throughout the RT-Benchmark collection. The entirety of the discussed features (and command-line options) are further discussed in Section 4.3 and exhaustively listed in the project documentation.

Table 2: List of metrics logged by RT-Bench and their units.

Metric	Description	Formula	Unit
`period_start`	Period start timestamp		ns and CPU clock-cycles
`period_end`	Period end timestamp		ns and CPU clock-cycles
`job_end`	Job end timestamp		ns and CPU clock-cycles
`deadline`	Absolute deadline timestamp		ns and CPU clock-cycles
`deadline_met`	Status of the deadline. 1 if met, 0 otherwise.		Boolean
`job_elapsed`	Absolute job response time	$job\_end - period\_start$	ns and CPU clock-cycles
`job_utilization`	Job utilization	$\frac{job\_elapsed}{period\_end - period\_start}$	Ratio
`job_density`	Job density	$\frac{job\_elapsed}{deadline - period\_start}$	Ratio
`l1_references`	L1 References (PMC)		Absolute number
`l1_refills`	L1 Refills (PMC)		Absolute number
`l2_references`	L2 References (PMC)		Absolute number
`l2_refills`	L2 Refills (PMC)		Absolute number
`inst_retired`	Instructions retired (PMC)		Absolute number

3.2 Measurements Processing

Alongside the mandatory core module, a.k.a. the RT-Benchmark Generator, the framework also includes a series of optional high-level scripts built on top of the generator. The provided scripts are written with high-abstraction-level languages such as *python* and *bash*. They aim to provide a well-rounded user experience in at least four ways: (1) they automatically perform common tasks such as empirically determining a benchmark's WSS, WCET, and ACET; (2) they ease the launch of interfering tasks, both memory- and CPU-intensive on both the same or other available CPUs; (3) they perform system-dependent preparation tasks such as migrating and pinning on selected CPUs to limit undesired interference; and (4) they generate plots of the obtained results using plotting libraries.

The script set is a prime example of tools exploiting the RT-Bench standard interfaces, setting the benchmark parameters following the standard command options, and extracting the measurements by parsing the reporting format.

3.3 Utils

The RT-Bench framework also comes with project maintenance and deployment tools, further improving portability and usability.

The framework provides a fully automated build system to generate RT-Bench benchmarks for each supported suite. It enables the building and management of suites individually and globally. Cross-platform compiling is supported for `ARM` and `x86_64`.

Additionally, complete documentation regarding the framework's RT-Benchmark generator is provided. This documentation is available in both HTML and LaTeX locally and on the framework's repository. It is generated by Doxygen [22] and already available online[4].

4 IMPLEMENTATION

This section presents the main technical details behind the implementation of our RT-Bench. This section focuses on the RT-Benchmark Generator, its mechanisms, and how it must be used to port a *generic* monolithic benchmark. Later in the section, the emphasis is put on the optional side tools offered with the framework to streamline common real-time oriented tests.

[4]See https://bastoni.gitlab.io/rt-bench/.

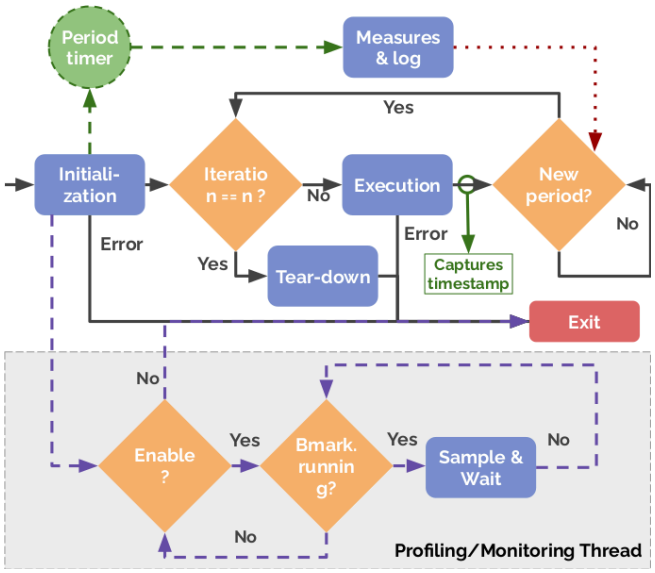


Figure 2: Flowchart of the mechanism used by RT-Bench.

As stated in earlier sections, RT-Bench has been designed with extensibility and portability in mind. This has led to some determining implementation decisions. The implementation presented in-depth in this section and evaluated in Section 5 assumes that Linux is the OS of reference. Even though most of the features only depend on POSIX, other features such as the available real-time scheduling policies are inherently dependent on the OS. Selecting Linux provides us with a sound selection of policies (e.g., `SCHED_FIFO`, `SCHED_DEADLINE`) and a simple interface to configure their parameters (i.e., `SYS_sched_*` syscalls).

4.1 Core Mechanism

At the heart of any of the benchmarks generated using RT-Bench lays the logic and mechanism in charge of enabling the desired features listed in Section 3. RT-Benchmark generator is responsible for invoking the entry points (to be implemented as part of the

porting of a new benchmark) at adequate moments. This enables RT-Bench to provide the features described in Section 3 to any compliant benchmark it is attached to.

A flow-graph representation of said logic is shown in Figure 2. The core logic is executed as a single-threaded process. The first step (or entry point) in the RT-Bench logic broadly consists of the **initialization** of the benchmarking environment. In addition to calling the associated benchmark's *Initialization* harnessing function, this initialization phase sets up every feature provided by RT-Bench using the user-specified inputs or the default ones. For instance, this includes the period, the deadline, or the amount of iterations. Noticeably, a timer-triggered (using a high-resolution timer) signal with the specified periodicity is set up. In Fig. 2, the timer is attached to the main thread, and its transitions are dashed and colored in green. At any point, if an error arises, a message is provided in output, and the benchmark is terminated (see `Exit` in Figure 2). Thereafter, the benchmark is ready to enter its periodic **execution** phase. The amount of iterations specified by the user (n) is enforced. There are two possible outcomes: the desired amount of iterations has been reached, or a few iterations remain to be performed. In the latter case, the benchmark is executed by calling the **Execution** harnessing function. Upon completion of the benchmark's workload execution, the process is blocked until a new period starts. In such case, the process loops back to the iteration comparison. Only iterations in which a job was started are considered in the comparison to make sure that n jobs are executed. Once all execution iterations have been performed, the benchmark can terminate gracefully by entering its *tear-down* phase, effectively calling the **Tear-down** harnessing functions.

Note that the *Initialization* and *Tear-down* phases are excluded from the measurements reported, preventing them from being tainted with extra noise from the setting-up and cleaning-up phases.

4.1.1 Measurements and Logging. The gathering and logging of the measurements at each period occur in two specific places: at periods' boundaries and after each execution phase.

Periods' boundary measurements are taken upon the reception of the period timer-triggered signal (*Measures & log* in Figure 2). The handling of the signal prompts the taking of the measurements and their logging. Once done, a new period is started by releasing the semaphore blocking the main execution thread (i.e., *the new period?* condition in Figure 2). This relationship is shown by the red dotted arrow in Figure 2. Note that this operation is only carried out if and only if the previous benchmark job has finished execution. Otherwise, the logging is filled with zeros instead, and no new jobs are released (job skipping). This also prevents logging irrelevant or misleading measurements. Deadline detection is carried out via a single boolean shared between the execution payload wrapper function and periodic signal handler. The flag is asserted when execution completes and de-asserted when logging completes.

4.1.2 Memory Footprint Watchdog. Upon request from the user via the provided command line options, a memory utilization watchdog is enabled through the alteration of memory allocation functions, namely, `malloc()`, `free()`, and `mmap()`. Following the framework scheme, the watchdog life-cycle is characterized by three phases: initialization, execution, and tear-down.

During initialization phase, the watchdog calls the `mallopt()` function in order to pre-allocate the user-specified amount of memory and disables the `mmap()` function. A preventive allocation of the requested memory space ensures that the allocated limit is never exceeded without requiring OS modifications. The functions `malloc()` and `mmap()` are wrapped such that, during the execution, any call to one of these two functions results in a working set size check. In case of failure, the benchmark is terminated. During the tear-down phase, the watchdog is disabled, meaning `mmap()` and `mallopt()` are re-enabled and their initial parameters are restored.

4.1.3 Memory Usage Profiling. If requested by the user via the command line options and available on the target platform[5], a thread in charge of monitoring the performance counters can be launched. In our case, the thread monitors and logs the performance counter associated with the L2 Refills. To mitigate the impact on the core logic thread, it is recommended to launch it on another core (see Section 4.3) and to reduce the *monitoring sample period*.

Unlike the core mechanism, the objective of this thread is to log measurements during the benchmark execution phases, instead of simply measuring before and after each execution. As shown in Figure 2, the thread is launched at the initialization phase and consists of a doubly-nested loop. The first step consists of a comparison, checking whether the thread is enabled by the main thread. The change of status is operated via a shared variable asserted at the initialization phase and de-asserted at the tear-down phase. In the latter case, it leads to the benchmark's termination. Otherwise, the thread enters a "sample-log-wait" loop as long as the benchmark remains in its execution phase (i.e., the *Bmark. running?* condition changes before and after the *Execution* block).

At the time of writing, the monitoring thread only samples the L2 Refills performance counter. This limitation is an implementation artifact that can be easily addressed. In addition, due to the close link between the performance counters and the platform implementing them, enabling support is not straightforward. The version of RT-Bench evaluated in Section 5 relies on the Linux Perf [2] and we specifically evaluate only events specific to ARM Cortex-A53 CPUs (Table 1).

4.2 Harnessing Functions and Extension

The capability of RT-bench to enable any benchmark with the set of desired features mentioned in Section 3 transparently is only possible if the benchmark has the three *harnessing points*.

From a practical point of view, the "adapted" benchmarks must implement these harnessing points. Each of them is a function with an immutable name and a clear objective. The *Initialization* harness point is implemented as the function `benchmark_init()`. It is in charge of initializing all the resources needed during the execution of the workload. Typically, memory allocation, variable initialization, and thread creation are carried out in the *Initialization* harness point. The *Execution* harness point is implemented as the function `benchmark_execution()`. As the name suggests, this function consists of the workload implementation that uses the previously-set variables. The *Tear-down* harness point is implemented as the function `benchmark_teardown()` and is the one in charge of freeing

[5] At the time of writing, only on ARM Cortex-A53

the resources used and, if desired, posting the obtained results. In other words, it ensures a clean termination.

Adapting an existing benchmark to RT-Bench requires the end-user to implement the three aforementioned harnessing functions, identify the relevant code segment corresponding to each harnessing function, and move the segments in the adequate functions. These alterations might seem heavy; however, in reality, most benchmarks already follow a form of setup-execute-teardown organization. Naturally, the initial organization of the benchmark to be adapted dictates the effort required. As an indication, we report on the changes and efforts required to adapt SD-VBS's `disparity` and `pca` via their number of changed lines of code using the cloc [1] tool. The `disparity` benchmark modifications amounts to 19 modified, 17 added and 12 removed SLOCs (or source lines of code) whereas modifications to `pca` amounts to 68 modified, 182 added and 216 removed SLOCs. Overall, the adaptation of the ten benchmarks composing the SD-VBS suite required a total of 302 modified, 925 added, and 689 removed SLOCs.

The end-user is free to define the content of the harnessing functions as desired. Nonetheless, when multi-threading is required, we recommend implementing the workload using the fork-join approach in each relevant function. In other words, we recommend that every thread created within a harnessing function is destroyed within the same function.

On the compilation side, an executable of the ported benchmark can be obtained via mainstream tools such as `gcc`. The ported source files shall not implement an entry function (i.e., `main`). Instead, the RT-Bench core interface must be linked in. As such, additional file directories must be added to the include path. This translates in utilizing `gcc`'s `-I/path/to/rt-bench/base/` option in addition to any benchmark-specific compilation flags and options.

4.3 Common Input Interface

As per the design goals presented in Section 3, any benchmark yielded by the RT-Benchmark Generator benefits from the same set of features, composing the homogenized input interface. Each of these features can be tailored via the enabled command-line options. The options under only represent a subset of the options made available to all benchmarks by RT-bench:

- `-p` : Relative period of a single benchmark execution;
- `-d` : Relative deadline of a single benchmark execution;
- `-l` : Log level;
- `-c` : Core affinity;
- `-f` : FIFO scheduling with specified priority;
- `-m` : Memory limit;
- `-t` : Number of tasks to execute before termination;
- `-b` : Benchmark specific arguments and options;
- `-P` : SCHED_DEADLINE period;
- `-D` : SCHED_DEADLINE deadline;
- `-T` : SCHED_DEADLINE runtime;
- `-M` : Enable PMC monitoring thread;
- `-C` : Core affinity for PMC monitoring thread;
- `-B` : PMC monitoring sample period;

The options listed above constitute the main options used in the Evaluation (see Section 5). An exhaustive list of the options, together with additional details, is provided in the project documentation.

4.4 High-level Automated Tests

The provided base scripts written in `Python3` constitute a collection of utility functions implementing profiling and real-time minded experiments. These tests have been used in the article's evaluation section (Section 5) to highlight the capability of the RT-Bench framework. At the time of writing, the set includes six experiments:

Minimum WSS. This test aims at empirically deriving the least amount of memory footprint required by the benchmark. To do so, the test explores the memory size allocation space via a binary search using the fact that the *memory watchdog* terminates any benchmark exceeding the user-defined allocation limited as the discriminant. I.e., if the program is terminated by the watchdog, the WSS is larger than the imposed limit; conversely, if the benchmark completed correctly, the WSS is smaller than (or equal to) the considered amount.

WCET. Thanks to the metrics reported by the RT-benchmarks, determining a benchmark's WCET can be empirically obtained via subsequent execution batches. The provided test *explores* candidate values by setting a default deadline value and *validates* by using the maximal observed execution time as the deadline for the following batch. The WCET is set as the minimum deadline value that reliably prevents misses when the benchmark is executed in isolation.

Schedulability Test. Based on previously established WCETs, this test looks at the rate of schedulable/unschedulable jobs as a function of the task's utilization. In this case, the deadlines are determined by dividing the previously-derived WCET by the target utilization. The test starts at 5% utilization and goes up to 100% in steps of 5%.

Caches Miss Rate. Leveraging the performance counters reported in the output interface, the Cache Miss Rate metric can be easily obtained by computing the ratio between the cache references and cache refills events. The test is applied on any available cache level.

Memory and CPU Intensity. This test investigates if a benchmark is CPU- or memory-bound by inspecting the ratio between the L2 cache misses and the number of retired instructions, two metrics natively reported by RT-Bench.

Memory Usage Profiling. Perhaps more importantly than knowing whether a benchmark is memory-bound, understanding the run-time demand is crucial for any system under memory bandwidth regulation. This test highlights the memory consumption phases a benchmark displays.

In any of the aforementioned tests, basic manipulations are performed. For instance, (1) before the benchmark execution, all tasks on the device are migrated to one core (often core 0) if requested by the user (2) measurements obtained are automatically plotted, and (3) co-running interference tasks are launched on other cores.

5 EVALUATION

This section showcases the capabilities and user-friendliness of the proposed framework, RT-Bench. The evaluation presented in this section consists in the set of experiments listed in Section 4.4: finding the minimum WSS (Section 5.1), determining the observed WCET (Section 5.2), performing the schedulability test (Section 5.3), studying the cache miss rates (Section 5.4), understanding whether

Table 3: Comparative table of the evaluation platforms

	Xilinx ZCU102	AMD RYZEN 9 5900HS
ISA	ARM64	x86_64
CPU	4×Cortex-A53 (@1.5GHz)	8×CPU (@3-4.6GHz)
L1	32KB+32KB I & D caches	8KB+8KB I & D caches
L2	1MB Unified cache	4MB Unified cache
L3	-	16MB Unified cache
DRAM	4GB DDR4	32GB DDR4
Linux	5.4.14	5.16.9
GCC	9.4.0	10.3

the benchmark is memory or CPU bound (Section 5.5), and observing the evolution of the memory consumption at run-time (Section 5.6).

The experiments have been performed on two different platforms: the Xilinx ZCU102 development board and the widely available AMD RYZEN 9 CPU model. Their architecture specifications and the version of the software tools (e.g., Linux kernel version and GCC version) are displayed in Table 3. From now on, the Xilinx ZCU102 is referred to as the "ARM platform" whereas the AMD RYZEN 9 is referred to as the "x86 platform".

Throughout the evaluation, the RT-Bench's capabilities are shown by using benchmarks issued from a RT-Bench adapted version of the San Diego Vision Suite (or SD-VBS) [38]. The exact benchmarks considered are `disparity`, `mser`, `localization`, `tracking`, and `sift`. In addition, all the available input sizes but `test`, `qcif`, and `full_hd` have been considered, that is `sim_fast`, `sim`, `sqcif`, `cif`, and `vga` (ordered by increasing size). For tests requiring an interfering co-runner, instances of the "bandwidth" benchmark issued from a RT-Bench adapted version of IsolBench [37] are launched. The interfering task instances sweep across a dedicated 100MB-wide buffer. Their number and their memory access mode (i.e., read or write) depend on the test performed and the platform capabilities.

In each experiment presented in this section, the benchmark under analysis is run using `SCHED_FIFO` and is assigned a priority of 99. Likewise, interfering co-runners are assigned a priority of 99. The *RT-Throttling* is turned off, allowing for a 100% CPU utilization.

5.1 Minimum Working Set Size Test

First, this experiment investigates the WSS of the supported SD-VBS benchmarks (Figure 3). Next, we place our emphasis on the WSS of `disparity` for all the available inputs (Figure 4). In both Figure 3 and 4 the minimal WSS found is reported by the height of the bars (y-axis in log scale). This set of experiments is only carried out on the x86 platform due to space constraints.

Figure 3 shows that, for the `vga` input, all the benchmarks require at least 10MB of main memory. Only `sift` and `localization` do not follow the rule as the former requires 100MB and the latter requires 1MB. However, as highlighted by Figure 4, the minimum required memory footprint is dependent on the input. In fact, one can observe that the WSS for a `vga` input is orders of magnitude bigger than that for a `sim_fast` input. Note that the observed size order matches the input size order.

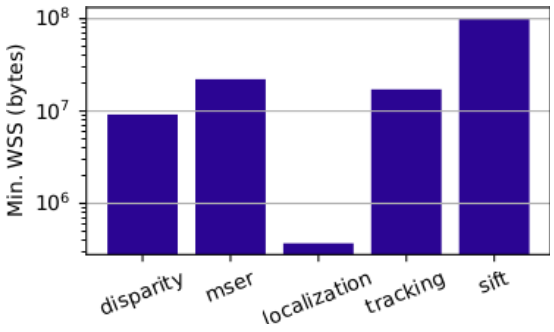


Figure 3: SD-VBS benchmarks minimum WSS for `vga` input.

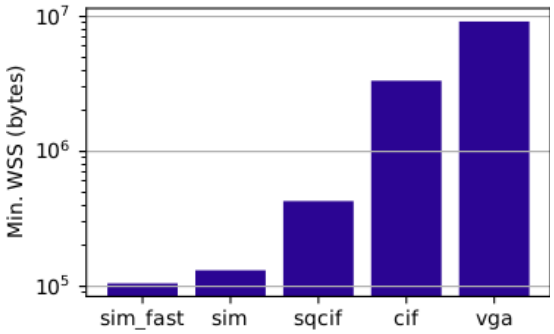


Figure 4: `Disparity`'s minimum WSS for different inputs.

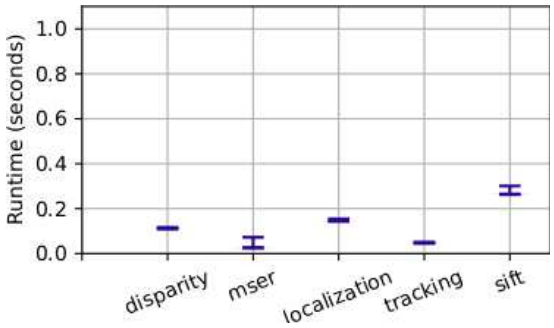
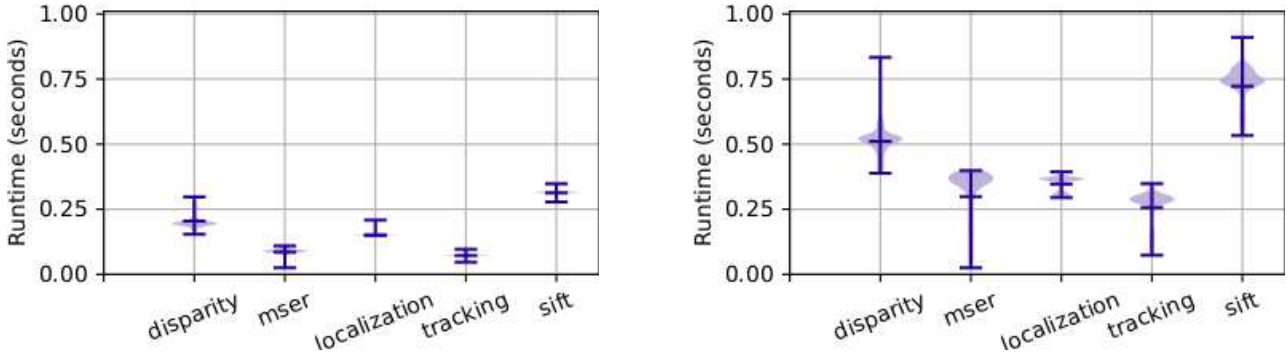


Figure 5: SD-VBS benchmarks WCET on x86 with `vga` input.

5.2 Worst case execution time test

In this experiment, the WCET test is used to understand the intrinsic behavior of the benchmarks when running in isolation (i.e., alone) and when they face memory interference from other cores. We present tests run on both the x86 and the ARM platforms.

To represent the distribution of the measured execution times, a violin plot was chosen. Each violin is associated with a benchmark running a `vga` input on the x-axis, and the y-axis reports their measured execution time in seconds. Each violin is composed of three horizontal lines representing the minimum, maximum and average measurements. The width of the violins represents the distributions of all the measurements.



(a) 4 cores interference.

(b) 6 cores interference.

Figure 6: SD-VBS benchmarks WCET tests on x86_64 on vga input with interference.

On the x86 platform, three scenarios are explored: (1) WCET in isolation (Figure 5), (2) WCET with 2 read and 2 write interfering cores (Figure 6a), and (3) WCET with 6 write interfering cores (Figure 6b). Figure 5 shows that without interfering processes, most of the benchmark execution samples do not present high variance. Conversely, `mser` and `sift` are the most likely to suffer from inter-core interference. This intuition is confirmed by Figure 6a which shows that, under interference, all benchmarks see their execution time distributions being stretched. In contrasts, Figure 6b shows that memory interference created by 6 cores writing data introduces higher variations in execution times. `Disparity` and `sift` are the most impacted with their WCET increased by twofold in Figure 6a, and eight-fold and threefold in Figure 6b, respectively.

On the ARM platform, two similar scenarios have been explored: WCET in isolation Figure 7a and WCET with 2 write-interfering cores Figure 7b. Unlike the x86 platform, the effect of interference creates a more consistent execution time distributions and only leads to longer execution times. However, as with the x86 scenarios, Figure 7a and 7b show that `disparity` and `sift` are the most impacted by interference.

5.3 Deadline Miss Ratio Test

To gain insight into the schedulability of the chosen benchmarks at a certain system load, two scenarios on the x86 platform and one scenario on the ARM platform are shown. On the x86 platform, Figure 8a shows the effect of two read and two write interfering cores, while Figure 8b shows the effect of six write interfering cores. On the ARM platform, there is only one scenario with two writing cores that generate interference, as shown by Figure 9. In both Figure 8 and Figure 9, the x-axis of the figures shows the utilization value, while the y-axis shows the number of benchmarks that met the deadline.

Figure 8a shows that only `mser` and `disparity` are severely impacted by the interference on the other four cores. While the impact on the other benchmarks is minimal. However, changing the interference pattern to six cores will severely impact all the benchmarks, keeping `mser` and `disparity` as the most impacted ones, as Figure 8b shows.

As Figure 9 shows, the ARM platform has a more predictable behavior than the x86 platform, having all the benchmarks meet the deadline or failing when the deadline gets too short to allow the benchmark to complete the execution with two writing cores that produce interference. As on the x86 scenarios, the most impacted benchmarks are `mser` and `disparity`.

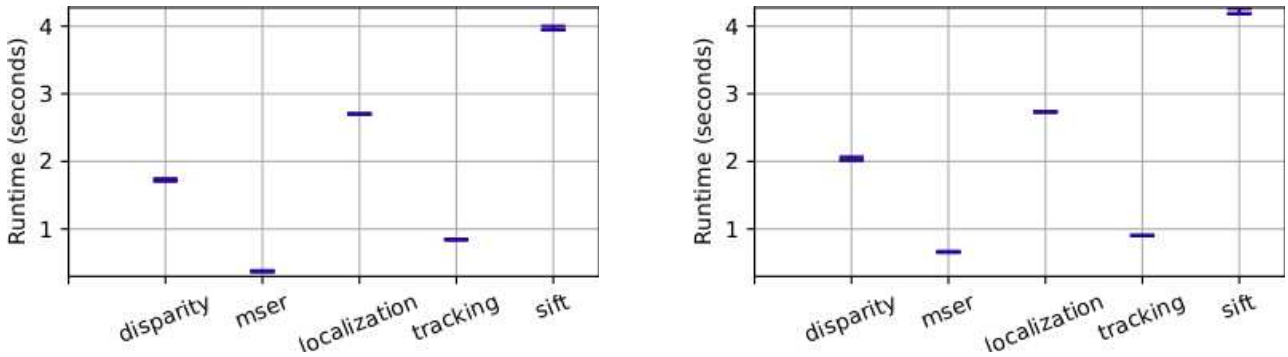
5.4 Caches Miss Rate

The cache miss rate experienced by a benchmark is a widely used metric to show how reliant on memory a benchmark is and the extent of memory interference impact. The L2 miss-rate experienced by the benchmarks running on the ARM platform is shown in Figure 10 (the bar clusters). In each bar cluster, the miss rate when running in isolation is drawn in blue (referred to as "solo"), whereas the observed miss rate under a two cores write contention is drawn in yellow (referred to as "interf"). Figure 10 highlights the existence of two categories. On the one hand, `disparity`, `sift`, and `tracking` are marginally impacted, hinting at a low temporal data locality (if the data is not reused later on, it does not matter whether it is evicted by an interfering task). On the other hand, `localization` and `mser` display a higher sensitivity to memory interference, hinting at a high temporal data locality. Remarkably, `mser` constitutes a hybrid case as it naturally displays a high miss rate in isolation and high sensitivity to memory interference.

5.5 Memory and CPU Intensity

To get further insight into the execution behavior of the chosen benchmarks, their ratio between the cache misses, and the number of instructions retired can be analyzed. This analysis is portrayed for the ARM platform by Figure 11, which shows, in a bar graph, the aforementioned ratio for all the benchmarks with and without interference.

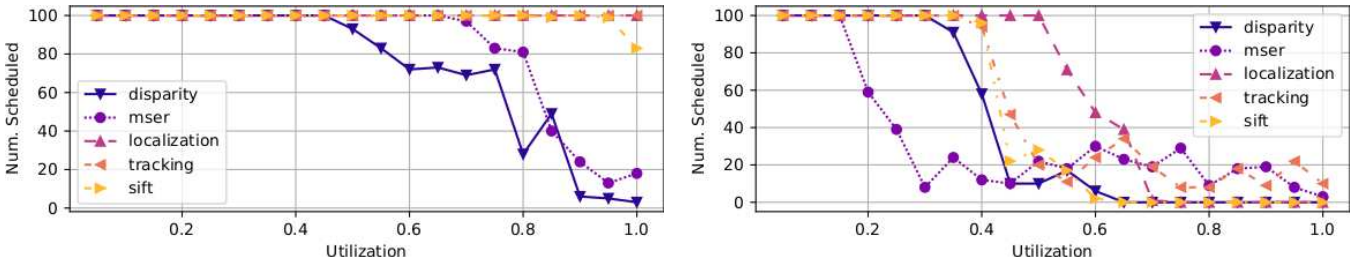
From Figure 11 it can be deducted if a benchmark is more memory or CPU bound. `localization` is an example of a CPU bound benchmark, while `disparity` is an example of a memory-bound benchmark. As for the previous test, the benchmark most impacted by interference is `mser`. Comparing Figure 10 and Figure 11 insight on how the cache misses affect the benchmark execution can be gained. While `sift` and `tracking` have more or less the same amount of cache misses, `sift` is more CPU bound than `tracking`, due to a smaller ratio between cache misses and instructions retired.



(a) no interference.

(b) 2 cores interference.

Figure 7: SD-VBS benchmarks WCET tests on ARM64 with vga input.



(a) 4 cores interference.

(b) 6 cores interference.

Figure 8: SD-VBS benchmarks schedulability tests on x86_64 on vga input with interference.

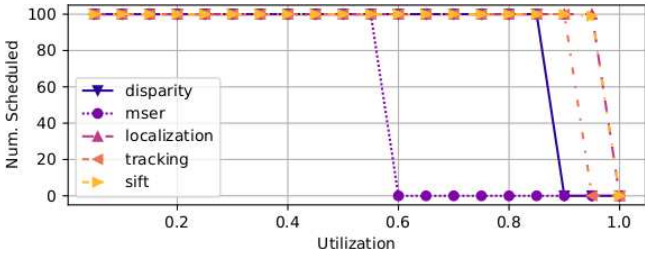


Figure 9: SD-VBS disparity schedulability test on ARM64 with vga input and 2 cores that produce interference.

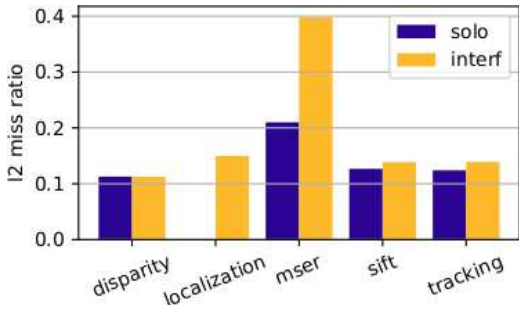


Figure 10: SD-VBS benchmarks' L2 cache miss-rate with and without interference (vga input).

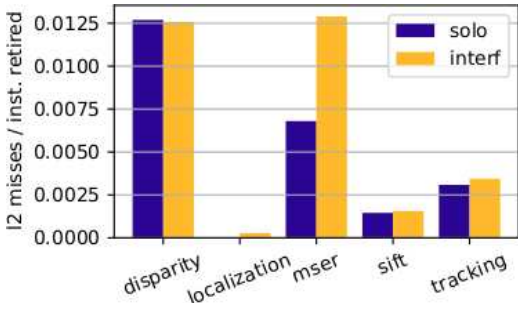
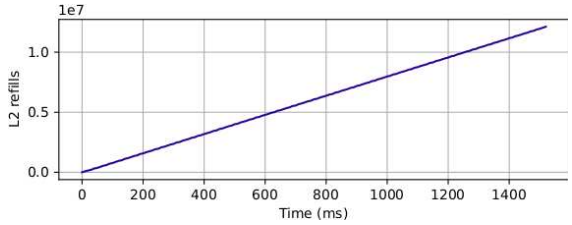


Figure 11: SD-VBS benchmarks' L2 cache miss-rate over instruction retired ratio with and without interference (vga input).

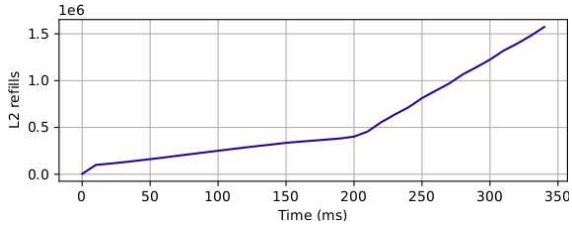
It can also be inferred that `localization` is the most CPU-bound benchmark, since it has the lowest ratio between cache misses and instructions retired.

5.6 Memory Usage Profile

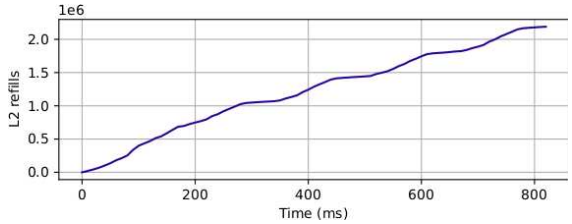
A memory usage profile can help identify how a benchmark uses memory during its execution. Figure 12 show the memory profiles of `disparity`, `mser` and `tracking` during their execution with a plot of the L2 cache misses on the y-axis and the time on the x-axis.



(a) SD-VBS `disparity`.



(b) SD-VBS `mser`.



(c) SD-VBS `tracking`.

Figure 12: The plot shows the total L2 cache refills during a single benchmark execution using the `vga` input.

Figure 12a shows that `disparity` is always accessing memory during its execution; this explains why it is so impacted by interference, even if it does not have a significant increase in cache misses when interfering processes are present. Instead, `mser` has three well-defined phases in which its memory access pattern changes significantly, explaining why it is impacted so heavily by interference in Figure 11. `tracking` has different phases with a different memory access pattern. This pattern highlights that it benefits from temporal and spatial locality in the data.

5.7 RT-Bench Framework Overhead

To quantify the overhead of the framework, an ad-hoc benchmark, `overhead`, has been used. The main payload of the `overhead` benchmark consists of an empty function. Therefore, the benchmark execution time will be dominated by the time spent to execute RT-Bench's core logic. Table 4 shows the measured overhead on the two platforms with and without using the performance counters. It can be observed that, on average, the overhead is contained between 20 to 50 μs.

6 CONCLUSION

The article presents RT-Bench, an open-source framework that aims to ease the tedious task of profiling and monitoring commonly used benchmark suites by providing a unified interface that can be built upon and re-used by the community. RT-Bench lays the foundation for a coherent benchmarking and profiling system for the real-time community. We provided an in-depth description of RT-Bench capability and outlined the main implemented features for a clean and reusable interface.

Through the evaluation of RT-Bench presented in Section 5 using well-known benchmarks suites such as SD-VBS and IsolBench, we showcase how the proposed implementation drastically simplifies the gathering and post-processing of experimental data.

While enabling the end-user with an interesting range of features, the presented version of RT-Bench is in its early days with a sizeable potential for community-fueled contributions and improvements. These include increasing the range of collected data, adding more performance counters, extending the provided benchmarks (including other popular suites like MiBench and TACLeBench), and extending the inputs to enable broader insight into the benchmark behavior with different inputs of the same size. Other possible avenues are the support for DAG tasks and a broader range of architectures, such as PowerPC and RISC-V. Finally, integration with IPC systems could be pursued to analyze inter-task dependencies.

Table 4: Rt-Bench overhead measurements

Platform	Min	Mean	Max	Std
x86 (μs)	5.49	24.57	515.99	13.99
ARM no Perf (μs)	1.36	30.63	95.91	8.39
ARM Perf (μs)	25.33	44.35	115.90	6.06
x86 (clock cycles)	18711	84987	1700193	47339
ARM no Perf (clock cycles)	2877	3086	9570	819
ARM Perf (clock cycles)	3817	4438	11588	605

ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. Andrea Bastoni and Denis Hoornaert were supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

REFERENCES

[1] [n.d.]. *cloc.* https://github.com/AlDanial/cloc
[2] [n.d.]. *Perf wiki.* https://perf.wiki.kernel.org/index.php/Main_Page
[3] [n.d.]. *RT-Test.* https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests
[4] [n.d.]. *RTEval.* https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rteval
[5] [n.d.]. *Splash2x benchmark suite.* https://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x
[6] Joshua Bakita, Shareef Ahmed, Sims Hill Osborne, Stephen Tang, Jingyuan Chen, F Donelson Smith, and James H Anderson. 2021. Simultaneous Multithreading in Mixed-Criticality Real-Time Systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 278–291.
[7] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, and Cong Liu. 2020. Co-Optimizing Performance and Memory Footprint Via Integrated CPU/GPU Memory Management, an Implementation on Autonomous Driving Platform. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 310–323. https://doi.org/10.1109/RTAS48715.2020.00007

[8] Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. 2020. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020-32nd Euromicro Conference on Real-Time Systems*. 1–22.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[11] Embedded Microprocessor Benchmark Consortium. [n.d.]. *EEMBC Benchmarks*. https://www.eembc.org/products

[12] Minyu Cui, Angeliki Kritikakou, Lei Mo, and Emmanuel Casseau. 2021. Fault-tolerant mapping of real-time parallel applications under multiple DVFS schemes. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 387–399.

[13] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIcs), Vol. 55)*, Martin Schoeberl (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.

[14] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2020. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 364–375. https://doi.org/10.1109/RTAS48715.2020.00011

[15] Golsana Ghaemi, Dharmesh Tarapore, and Renato Mancuso. 2021. Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:25. https://doi.org/10.4230/LIPIcs.ECRTS.2021.4

[16] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. 2021. DNA: Dynamic Resource Allocation for Soft Real-Time Multicore Systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 196–209.

[17] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[18] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14.

[19] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. 2017. Waters industrial challenge 2017. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. https://waters2017.inria.fr/

[20] Mohamed Hassan. 2020. Discriminative Coherence: Balancing Performance and Latency Bounds in Data-Sharing Multi-Core Real-Time Systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*, Marcus Völp (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:24. https://doi.org/10.4230/LIPIcs.ECRTS.2020.16

[21] Mohamed Hassan and Rodolfo Pellizzoni. 2020. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*, Marcus Völp (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:24. https://doi.org/10.4230/LIPIcs.ECRTS.2020.23

[22] Dimitri van Heesch. 2021. *Doxygen*. Dimitri van Heesch. https://www.doxygen.nl

[23] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. 2021. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:22. https://doi.org/10.4230/LIPIcs.ECRTS.2021.2

[24] Bashima Islam and Shahriar Nirjon. 2020. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 95–109.

[25] Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. 2021. Safety-Aware Integration of Hardware-Assisted Program Tracing in Mixed-Criticality Systems for Security Monitoring. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 292–305.

[26] Anirudh Mohan Kaushik and Hiren Patel. 2021. A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 105–117.

[27] Filip Marković, Jan Carlson, Sebastian Altmeyer, and Radu Dobrin. 2020. Improving the accuracy of cache-aware response time analysis using preemption partitioning. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[28] Filip Marković, Jan Carlson, and Radu Dobrin. 2020. Cache-aware response time analysis for real-time tasks with fixed preemption points. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 30–42.

[29] Filip Marković, Jan Carlson, Sebastian Altmeyer, and Radu Dobrin. 2020. Improving the Accuracy of Cache-Aware Response Time Analysis Using Preemption Partitioning. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 165)*, Marcus Völp (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.5

[30] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. 2006. Papabench: a free real-time benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[31] Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. 2020. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 213–225.

[32] Shahin Roozkhosh and Renato Mancuso. 2020. The potential of programmable logic in the middle: cache bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 296–309.

[33] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 101–111.

[34] Lui Sha, Tarek Abdelzaher, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysius K Mok, et al. 2004. Real time scheduling theory: A historical perspective. *Real-time systems* 28, 2 (2004), 101–155.

[35] Dharmesh Tarapore, Shahin Roozhkhosh, Steven Brzozowski, and Renato Mancuso. 2021. Observing the invisible: Live cache inspection for high-performance embedded systems. *IEEE Trans. Comput.* (2021).

[36] Corey Tessler, Venkata P Modekurthy, Nathan Fisher, and Abusayeed Saifullah. 2020. Bringing inter-thread cache benefits to federated scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 281–295.

[37] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461361

[38] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 55–64. https://doi.org/10.1109/IISWC.2009.5306794

[39] Corinna Vinschen and Jeff Johnston. [n.d.]. *Newlib*. https://sourceware.org/newlib/

[40] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.

[41] Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn, and Hiren Patel. 2021. A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 92–104.