RT-Bench: A Long Overdue Update

Mattia Nicolella
Boston University
Boston, Massachusetts, U.S.A.
mnico@bu.edu

Denis Hoornaert
Technical University of Munich
Munich, Germany
denis.hoornaert@tum.de

Renato Mancuso
Boston University
Boston, Massachusetts, U.S.A.
rmancuso@bu.edu

***Abstract*—RT-Bench is a framework and community project that aims to establish a unified set of benchmarks with a homogeneous launch and result reporting interface, and with a simple build system. RT-Bench targets academic researchers and industry practitioners interested in understanding the performance characteristics of embedded/real-time systems when tested over realistic use-case applications. To facilitate real-time systems research, RT-Bench is designed from the ground up to include a set of fundamental capabilities such as periodic execution, selectable OS scheduler, and native and multi-architecture performance counters support, to name a few.**

RT-Bench has undergone continuous improvements and extensions. This paper reviews the most recent additions and features of the framework. Most prominently, these include heap migration, synchronized benchmark release, and experimental support for multi-threaded applications. This contribution includes a tutorial session with template benchmarks to showcase the new features and illustrate the process of integrating new benchmark suites.

***Index Terms*—Benchmarking, Real-time, Profiling, Periodic benchmarks**

I. Introduction

For practitioners and academics, benchmarking constitutes an essential step in testing and validating their systems regardless of the application domain. Naturally, many domain-specific benchmark suites have emerged over time as a result of independent efforts. On the one hand, this organic growth has offered a broad and diversified portfolio of benchmarks. On the other hand, however, it has caused an inherent fragmentation w.r.t. the set of features, launch/command interfaces, and supported metrics. This lack of a *de facto* standard primarily hinders reproducibility, as well as productivity and adoption, since practitioners must manually adapt each suite of interest to their needs via a repetitive and time-consuming process.

Since its release [1], RT-Bench [2] has aimed to reduce these frictions through a rich standardized interface that enables compatible benchmarks to tap into its feature set seamlessly. For instance, the framework allows users to swiftly leverage common timing features, such as periodic execution and reporting of elapsed time, in all the included benchmarks. Not only does RT-Bench focus on features relevant to real-time system benchmarking, but it also provides native support for resource profiling, which has proven helpful in understanding run-time resource requirements and pinpointing performance bottlenecks in more general settings.

The framework also strives to combine user-friendliness, requiring minimal effort when adapting to new benchmarks. To that end, RT-Bench's contributors have maintained an extensive documentation [3] of the framework, ranging from the supported benchmark suites to the framework's APIs.

This article and its associated tutorial session aim to reiterate RT-Bench's core concepts, mechanisms, and goals while formally introducing the newly ratified features.

II. What's new?

Since its first release in 2022 [1], RT-Bench has known a relative success within the real-time community [4]–[10][1]; calling for many improvements, feature additions, feature revision, bug fixes and code consolidation. This section describes the most prominent changes brought to RT-Bench.

A. Continuous Back-to-back Executions.

One of the first features introduced in RT-Bench in an effort to more realistically reflect the behavior of real-time applications was the enforcement of periodic execution. As RT-Bench increased in adoption, however, it was brought to our attention that many scenarios and use cases exist where this mode of operation is not ideal and, in fact, undesirable.

Typically, these correspond to cases where the role of the benchmark is to create pressure on the system's resources (i.e., bandwidth from IsolBench [11]). As such, the intermittent activity of interfering workloads that comes from periodic releases and missed deadlines (and consequent job skipping, showed in Fig. 1, top part) creates *"pressure gaps"* that can make empirical *worst-cases* harder to observe and reliably reproduce. Such gaps are also inconvenient when attempting to collect stable readings of the performance counters.

Hence, a *"continuous back-to-back execution"* mode has been introduced (Fig. 1, bottom part). It can be enabled simply by omitting a period (`-p`), ensuring retro-compatibility with existing RT-Bench command options. Note that nothing else changes as the benchmark execution routine is still executed in a loop until a `SIGINT` is received or until the specified number of tasks instances (`-t`) has been completed.

B. Heap Migration

The push toward heterogeneous System-on-Chips is not just confined to processing elements. Modern embedded platforms, in particular, also tend to feature a diversified array of on- and off-chip memories with different sizes and temporal

[1]For sake of transparency, cited papers exclude research items involving the authors of the original paper [1] and their close collaborators.

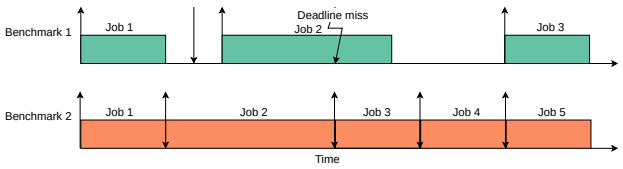

Fig. 1: RT-Bench execution modes: Periodic execution (top) and continuous back-to-back (bottom).

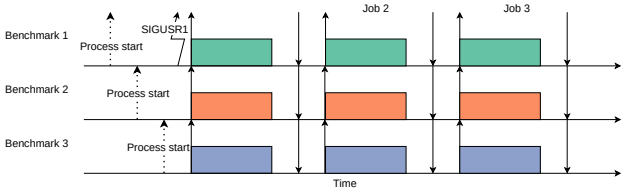

Fig. 2: RT-Bench synchronized release example with three benchmarks.

characteristics. On Linux-based systems, these memories are often treated as reserved memory nodes. Thus, unless system designers add ad-hoc support, these nodes remain natively inaccessible to standard user-space applications.

To enable the study of benchmarks' memory affinity, RT-Bench integrates a user-friendly *heap migration* mechanism. When heap migration is enabled, RT-Bench switches to a custom heap allocator over an internally allocated memory region. The user can control the type of allocation via command-line parameters. Specifically, when enabling heap migration (`-H`), users can indicate the target memory in two ways: (1) by providing a physical memory address; or (2) by providing a file pathname.

In the former case, RT-Bench will perform a non-cacheable memory mapping using `/dev/mem` and will use said mapping as the applications' heap memory region. In the latter case, RT-Bench will memory-map the provided file. If this is a normal file (e.g., in a `tempfs` filesystem), the resulting memory region will be cacheable. This option is handy as reserved heterogeneous memory nodes can be exposed to user space by exporting them as block devices that can be memory-mapped. Kernel-level support for the creation of said interface, however, is out of the scope of RT-Bench. When using the (`-H`) option in either mode, the end-users are responsible for providing adequate access to the underlying memory (e.g., by ensuring it is a usable physical address/file) to avoid bus errors.

Note that a heap size limit parameter (`-m`) is required because RT-Bench must terminate the target benchmark if the working set size exceeds the size of target memory region.

C. External Synchronized Release

As RT-Bench runs atop of a rich full-fledged operating system (i.e., Linux), running several benchmarks concurrently to observe and study their interactions is already possible. However, no control over the release time of the workloads is usually possible. This makes the simulation of specific conditions, such as the worst-case job release pattern, hard to perform.

Thus, an initial release (offset) synchronization mechanism has been added to RT-Bench. This mechanism, shown in Fig. 2, allows different benchmarks or different instances of the same benchmark to be released together via the newly introduced (`-s[=TASKSET_NAME]`) parameter. When the parameter is used, the released benchmarks behave as if they belong to the same task set with synchronized release offset. Specifying a task set name is optional. If an explicit name is not provided, RT-Bench will default to an implicit name.

Practically speaking, RT-Bench's synchronization unrolls in three phases. Initially, all workloads initialize themselves until they are ready to execute for a first iteration. There, they wait to receive a synchronization signal (i.e., `SIGUSR1`). This behavior allows users to add an arbitrary number of benchmarks to the set by simply starting them with the same (`-s`) parameter value. Once all the desired benchmarks have been launched, the user can start the full experiment by sending the `SIGUSR1` signal to any one of the benchmarks in the set. Once any of the benchmarks receives the signal, said benchmark becomes the *synchronization manager*. The manager is endowed with the responsibility to (1) determine a synchronized release time in the near future and (2) send a signal to wake up all the other benchmarks (subordinates). Upon reception of the signal, each subordinate benchmark, as well as the manager, will configure an absolute release timer with the exact coordinate time determined earlier.

Note that the mechanism still involves the end-users as they must manually start the required benchmarks and choose a unique task set name (`TASKSET_NAME`). They also must send the first `SIGUSR1` signal once all the desired benchmarks have been launched. Alternatively, a `synch-helper` tool is also provided which allows users to specify how many benchmarks to wait for before the start `SIGUSR1` signal is sent.

D. Extended Reporting of Benchmark-specific Metrics

For some benchmarks, the default performance metrics reported by RT-Bench are only a subset of the information useful for the experiment at hand. Since its initial release, RT-Bench offers the possibility to report one benchmark-specific metric for each benchmark. This feature is used, by the applications in the IsolBench suite [11], which also report measured memory latencies and bandwidths. However, in a number of cases, limiting benchmark-specific metric reporting to a single value was deemed too restrictive. For instance, a benchmark performing neural network inference for object identification might want to simultaneously report the number of detected objects *and* the corresponding confidence scores.

For this reason, the benchmark-specific reporting mechanism has been revamped. In this revision, benchmark-specific metrics and metadata are stored in a struct named `extra_measurement` aggregating pairs of headers and metrics. At run-time, the struct is populated by the user-

defined `benchmark_log_data()` routine with the desired metrics and their human-readable name. Accordingly, the `EXTENDED_REPORT` compilation flag has been deprecated.

E. Performance Monitoring Thread Improvements

Recall that RT-Bench included the option to spawn a performance monitoring thread *PMThread*. The PMThread runs in parallel with a given benchmark, and its goal is to perform high-frequency sampling of architectural performance counters to more accurately profile the benchmark under analysis.

An interesting challenge in the design of the PMThread is how to store the trace of performance samples efficiently. In the initial version, sample measurements[2] were stored in one contiguous container (i.e., a buffer) where each entry corresponded to a timestamp in the execution. For a time bucket Δ_t, the i^{th} entry contains the performance measurements sampled at instant $i \times \Delta_t$ of the benchmark run-time. Such pre-allocated container would limit the execution time of the benchmark and eventually lead to a buffer overflow.

The reworked implementation, instead, features support for a list of buffers that can hold these measurements. This is achieved dynamically during the execution phase as follows. Whenever a buffer is filled, a new buffer is allocated and added to the list, with a size that is twice as large as the previous buffer in the list. The approach limits the number of memory allocations during execution while not copying/moving any previously acquired data samples. Finally, any memory used to hold performance counters is not tracked by the memory watchdog, and thus, it does not count against the benchmark's memory limit (`-m`).

F. Experimental Multi-thread Support

The feedback following the previous release of RT-Bench lamented the lack of support for multithreading. The absence was justified by several—feasible but tricky—implementation hurdles such as synchronization and core affinity assignment.

Since then, steady progress has led to the introduction of an experimental support for multithreading. It is now possible to easily use the framework with a multithreaded benchmark since new APIs to (1) spawn new threads and (2) define a parallel computation section with these threads are available. With these APIs the main execution thread can operate before, during and after the parallel section; with minimal changes to RT-Bench's execution logic as shown in Fig. 3.

Note that, at the time of writing, multi-threading cannot be used in conjunction with other options. For instance, as the subsystem used for heap migration and Working Set Size (WSS) reports are not thread-safe, multi-threading can only be used provided no dynamic memory allocations are performed during the execution phases. Likewise, performance counter reports generated by the PMThread only cover the main thread.

III. Demonstration session outline

This section provides a step-by-step overview of the tutorial session accompanying this paper. The objective is to demonstrate the user-friendliness of RT-Bench and provide the audience with key pointers to get started with their projects.

1. Building and Launching a Benchmark. Using a project template [12] specifically made for this occasion, we will showcase how RT-Bench-adapted benchmarks can be compiled and executed. In particular, we will focus on the required command line parameters, scenario configuration via `.json` files, and where to find built binaries. This part will cover how to release a taskset, a heap migration example, and how to retrieve performance measurements, including an example using the PMThread.

2. Adding a Benchmark. This part will showcase how to extend RT-Bench with additional benchmarks [13], whether single- or multi-threaded. The inspection of the single-threaded benchmark from the template will serve as a stepping stone to revise RT-Bench's basics. Specifically, it will include how to split a `main` function into the three required harness functions (i.e., `init`, `execute`, and `teardown`) and tips on how to manage any inputs modified in place during execution. This part of the demo will also cover how to add custom logged metrics. Finally, the steps that must be undertaken to tap into the *experimental* multi-thread support will be shown by adapting the benchmark at hand.

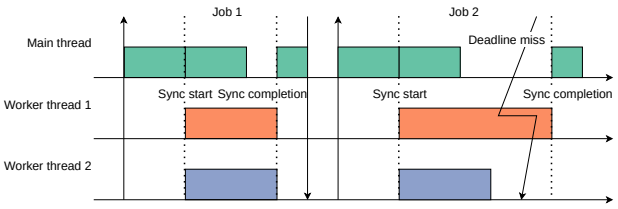

Fig. 3: Execution model for multithreaded benchmarks.

IV. Conclusion & Future Work

RT-Bench benefits from a continued effort in maintaining the project and adding new features. Future work is directed towards (1) adding multithreaded benchmarks and improving the support for multithread execution, (2) supporting C++ benchmarks, and (3) the ability to define and release a taskset from a single `.json` specification. Other future work includes the construction of a RT-Bench database, which contains metrics of different benchmarks categorized by platform.

The authors are certainly committed to providing community support, maintaining, and extending the RT-Bench project. Nonetheless, they also welcome inputs and contributions by the community, such as new benchmarks, features, and support for additional platforms: these are fundamental to keep the project relevant, up to date, and ultimately useful.

V. Acknowledgments

This research was supported by the National Science Foundation (NSF) under grant number CSR-2238476. Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

[2] i.e., a `struct` aggregating a selected set of PMCs such as `L2_refills`.

REFERENCES

[1] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "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*, 2022, pp. 184–195.

[2] Rt-bench repository. [Online]. Available: https://gitlab.com/rt-bench/rt-bench/-/tree/OSPERT25

[3] Rt-bench documentation. [Online]. Available: https://rt-bench.gitlab.io/rt-bench/branch/OSPERT25/index.html

[4] C.-F. Yang and Y. Shinjo, "Compounded real-time operating systems for rich real-time applications," *IEEE Access*, vol. 13, pp. 26 079–26 104, 2025.

[5] A. Oliveira, G. Moreira, D. Costa, S. Pinto, and T. Gomes, "IA&AI: interference analysis in multi-core embedded AI systems," in *Data Science and Artificial Intelligence*, C. Anutariya, M. M. Bonsangue, E. Budhiarti-Nababan, and O. S. Sitompul, Eds. Singapore: Springer Nature Singapore, 2025, pp. 181–193.

[6] M. A. Soomro, A. Nasrullah, and F. Anwar, "Poster abstract: Time attacks using kernel vulnerabilities," in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 626–627. [Online]. Available: https://doi.org/10.1145/3715014.3724040

[7] M. A. Khelassi and Y. Abdeddaïm, "Impact of compilation optimization levels on execution time variability," in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2024, pp. 1–4.

[8] M. A. Khelassi, "Using statistical methods to model and estimate the time variability of programs executed on multicore architectures," Theses, Université Gustave Eiffel, Dec. 2024. [Online]. Available: https://hal.science/tel-04921969

[9] W. Dewit, A. Paolillo, and J. Goossens, "A preliminary assessment of the real-time capabilities of real-time Linux on Raspberry Pi 5," in *18th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*. Alexander Zuepke and Kuan-Hsun Chen, 2024.

[10] K. Hosseini, "Real-time system benchmarking with embedded Linux and RT-Linux on a multi-core hardware platform," Master's thesis, Linköping University, Department of Computer and Information Science, 2024.

[11] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

[12] Rt-bench benchmark suite template. [Online]. Available: https://gitlab.com/rt-bench/templates/benchmark-template/-/tree/OSPERT25

[13] Benchmark template to extend rt-bench. [Online]. Available: https://gitlab.com/rt-bench/templates/adapt-bmark/-/tree/OSPERT25