Work in Progress: KDBench - towards open source benchmarks for measurement-based multicore WCET estimators

Marwan Wehaiba El Khazen* †, Kevin Zagalo*, Hadrien Clarke*, Mehdi Mezouak*, Yasmina Abdeddaïm‡ *, Avner Bar-Hen§ *, Slim Ben Amor†, Rihab Bennour†, Adriana Gogonel†, Kossivi Kougblenou†, Yves Sorel* and Liliana Cucu-Grosjean*
*Inria, France, Email: firstname.lastname@inria.fr
†StatInf, France, Email: firstname.lastname@statinf.fr
‡UGE, ESIEE, France, Email: firstname.lastname@esiee.fr
§CNAM, France, Email: firstname.lastname@cnam.fr

***Abstract*—The real-time systems community is facing the lack of benchmarks adapted to measurement-based worst-case execution time (WCET) estimators. We provide in this paper first steps towards such benchmarks by proposing them for single core microcontrollers, while we leave as future work the migration to multicore microcontrollers. The considered benchmarks are the programs of an open source drone autopilot. We conclude the paper by underlining the main difficulties of such migration.**

***Index Terms*—measurement-based WCET estimation, benchmarks**

I. Motivation and definitions

The real-time community is facing the lack of benchmarks adapted to measurement-based analyses. Existing benchmarks for the estimation of WCET [1]–[3] have been proposed to estimate WCETs mainly for static analyses. They contain simple programs that are not accompanied by a measurement protocol. They do not take into account functional dependencies between several benchmarks like those due to shared global variables which, of course, may influence their execution times. On the other hand, measurement-based analyses require execution times measured while executing programs on microcontrollers, similar to those used in the embedded systems industry.

Our contribution is the proposition of measurement-based benchmarks, KDBench, for real-time embedded systems[1]. More formally, we understand by measurement-based benchmarks a 4-uple (A, p, $\mathcal{M}$, $c(A)$) composed by a program A, a microcontroller p, a measurement protocol $\mathcal{M}$ and an ordered sequence of execution times $c(A)$. For the program A, one may provide the source code as well as the binary code. A measurement protocol $\mathcal{M}$ may be defined by the variation of the input variables (associated to sensors) of these benchmarks. In our case, the variation of the input variables is obtained by collecting them during a simulated flight. The fourth component, the ordered set of execution times, $c(A)$ is proposed to overcome the difficulty of the reproducibility of results [4] while comparing execution times measured for the same program on slightly different microcontroller configurations. Moreover, we provide these measurements by using information collected at the scheduler level, thus the impact of the measurement protocol is negligible on the variation of measured execution times. Last, but not least this fourth component improves the access of our community to hardware-in-the-loop (HitL) benchmarks. We understand by HitL that the execution of the benchmarks is done on a microcontroller while sensors and actuators of a considered cyber-physical system (CPS), as well as its physical environment, are simulated. Indeed, our community does not often provide numerical results for programs executed on microcontrollers because of the important effort of implementation required for such execution, or the lack of access to these processors. This may prevent the community in proposing realistic models describing the impact of existing microntrollers and thus propose results that may be not realistic w.r.t. the microntrollers used by the real-time industry.

The ROSACE case study [5] contains the closest benchmark programs w.r.t our following objectives: open-source and executable on multicore processors. Nevertheless, to the best of our knowledge, the ROSACE programs are not implemented on a processor hosted on a board with sensors and actuators. In order to complete our benchmark proposal, we would like to provide to users the variation of input variables as well as output variables of programs. The PX4 autopilot programs fulfill this requirement, without an expected important development effort from our side. Indeed, the PX4 programs are proposed after testing their implementation w.r.t real flights implying that stable configurations of periods exist.

With respect to the WCET estimation problem, we consider in this paper statistical estimators to illustrate the measurement-based WCET analyses, but one may use any existing measurement-based or hybrid WCET analysis [6].

The paper is organized as follows. We describe in Sec-

This research is partially funded by the FR PSPC STARTREC and the FR Inria ADT grant KDBench.

[1]All information related to these benchmarks are available at https://team.inria.fr/kopernic/kdbench

tion II the open source programs of a drone autopilot that we have considered as target for measurement-based benchmarks. Their transformation in benchmarks for real-time embedded systems is described in Section III. In Section IV we present first results on the single core benchmarks, while our target multicore benchmarks are presented as future work in Section V.

II. Drone autopilot PX4 limitations for a direct use as benchmark for WCET estimators

The PX4 autopilot is an open source flight control software [7] designed by ETH Zurich. PX4 runs on top of NuttX, a Unix-like OS developed by Gregory Nutt [8] or on top of Linux, both compliant with POSIX.

The PX4 program is written in C++ and contains two main parts. The PX4 Middleware provides an infrastructure for internal communications among all programs, called modules afterward, via the micro Object Request Broker, called uORB, using a publish/subscribe mechanism, and for external communications between PX4 and offboards applications like the ground control station (GCS) using the standard protocol MAVLink [9]. The PX4 Flight Stack contains a lot of modules implementing drivers, control algorithms, filters, etc., for manual and autonomous flight missions. In addition with the POSIX compliance, all these features are a pledge for the portability of PX4. Each module is a NuttX task started at the beginning of the PX4 program. It is composed of an infinite loop which performs the following sequence of actions: subscribe to one or more topics via uORB, wait for new data produced by other modules using a poll function (blocking wait), read data from corresponding topics via uORB when they are available, compute data of the module and, finally, publish the resulting data to uORB. Actually, the Flight Stack is a data precedence graph where modules are vertices of the graph and their data precedence relations are edges of the graph, starting from the hardware sensors which are inputs of the autopilot down to the motors which are outputs of the autopilot. The modules read data from sensors, estimate the position and the attitude using a Kalman filter (EKF2), control the position (Position) and the attitude (Attitude) of the drone, handle the navigation (Navigator), manage the state of the drone (Commander), and write data to actuators, i.e., motors. Every module waits for data produced by the previous modules to which it is related to, according to the data precedence graph of modules and, thus, according to the subscribed topics. Then, it performs some computations with these data and publishes other data for other modules. Every module is an infinite loop composed of a sequence of actions and one of these actions waits through a poll function, being blocked until new data, produced by other modules, are available.

Every module waiting for the availability of data may have a variable period, such that it is not possible to guarantee that a module is sporadic or periodic. Moreover, unavailable data may lead to variable execution times. Both important variations in terms of periods and execution times may increase the difficulty of schedulability analyses.

In order to ensure that the schedulability analyses are possible on systems integrating PX4 autopilot modules, we transform the graph of modules corresponding to the PX4 program into a graph of real-time tasks.

III. Real-time autopilot PX4-RT

We transform the PX4 program by modifying every module. First of all, we guarantee that every module is executed periodically at an accurate period. This is achieved by using the C++ function `hrt_call_every(delay,interval,callout,arg)` provided by PX4 libraries. This function, called ”HRT” afterwards, takes benefit of the high resolution timers of the microcontroller. It calls, the first time, the function `callout(arg)` after a certain delay `delay` and, then, repetitively calls the function `callout(arg)` every interval of time `interval`. More precisely, after the delay `delay` it loads the value `interval`, corresponding to the desired period, in a high resolution timer which is decremented until it reaches zero. At this instant the timer is reloaded with the value `interval` and, then, an interruption routine corresponding to `callout(arg)` is triggered. In our case `callout(arg)` only unblocks a semaphore. Thus, an HRT and a semaphore are associated to every module, such that HRT unblocks periodically the associated semaphore at the specified period.

Identically to a module, as explained in Section II, every real-time task is a NuttX task that is started at the beginning of the PX4 program. But now, its first action consists in executing HRT which unblocks periodically the associated semaphore at the specified period, followed by an infinite loop which performs the following sequence of actions: subscribe to one or more topics via uORB, wait until the semaphore is unblocked and, then, blocks the semaphore, read data from corresponding topics via uORB, compute data of the task, publish the resulting data to uORB. Consequently, since there is no more poll functions that wait for available data, the latter are read at the period of the task. This makes possible that data have several times the same value, but data are never lost. Even though this approach seems less smart than the one consisting in waiting for available data, executing the control tasks at an accurate period guarantees that data they compute are transmitted with the minimum jitter from one task to another task, and finally to the actuators, which is of crucial importance for accurately controlling the motors of the drone. If a task has a state, e.g., EKF2, it is not correct in term of control to use twice the same data. Therefore, when such a task reads a data which is not available, instead of executing normal computations, we execute an empty loop with the same duration as the normal computations. Therefore, every real-time task is executed accurately at its period that the user can change when testing. The task reads data from the subscribed topic of the corresponding module, and publishes data to other topics. Thus, we obtain a graph of dependent real-time tasks and we illustrate a simplified version of this graph in Figure 1, where the main real-time tasks are depicted in green. We qualify as ”real-time” this new version of PX4 and name it PX4-RT.

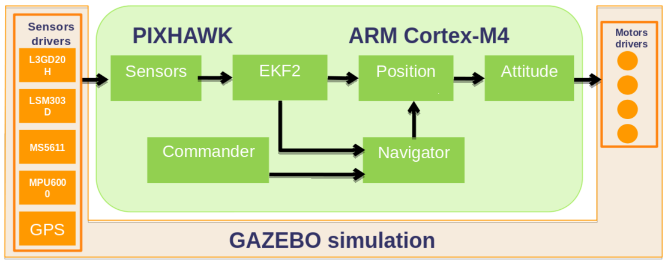

Fig. 1. Graph of dependent real-time tasks and HitL Simulation

In order to collect execution times of tasks, we execute a large number of times a given autonomous flight mission, i.e., with the same trajectory of GPS points, with PX4-RT running on the single core ARM Cortex-M4 included on the Pixhawk board, the most common board for autopilot PX4-based drones. We measure the execution times inside a HitL simulation based on the widely-used simulator Gazebo [10] (see Figure 1) providing a large amount of execution times that are extremely time consuming to obtain during real flight missions. The modules of PX4 or the real-time tasks of PX4-RT are scheduled by the scheduler of NuttX, according to a fixed priority FIFO, i.e., when several modules have the same priority they are scheduled FIFO. We use a Rate Monotonic [11] priority assignment, where every task has a period inherited by every module in PX4. We have 9 real-time tasks in the graph of dependent tasks, their list is detailed in Table I, as well as their periods of activation. They are ordered according their priority from the highest priority (on the left of the table) to the lowest priority (on the left of the table).

modules	snsr	rate	ekf2	actl	pctl	fmgr	hte	navr	cmdr
T (ms)	3	4	4	5	5	6	7	50	100

TABLE I
LIST OF THE REAL-TIME TASKS WITH THEIR ASSOCIATED PERIODS

IV. PROPOSED SINGLE CORE BENCHMARKS FOR MEASUREMENT-BASED WCET ESTIMATORS

Our single core benchmark is composed of the real-time tasks (programs) of the PX4-RT autopilot executed on an ARM Cortex-M4. As this microcontroller has no cache memory, nor any hardware acceleration feature, then the status of the microcontroller features has no impact on the variation of the execution times.

For each program of the PX4-RT autopilot, we provide the following components as part of the benchmark KDBench: the source code of all autopilot programs A, made publicly available on a gitlab under an open source licence; the description of the microcontroller p which is, in our, case, the ARM Cortex-M4; a measurement protocol $\mathcal{M}$ described by an ordered sequence of values for input variables collected during a recorded flight (see Figure 3); an ordered set $c(A), \forall A$ of execution times as well (Figure 4).

Other possible utilization of the KDBench are probabilistic scheduling analyses [12], [13]. Within our framework, we provide an overview of response times, execution times and release jitters distributions for each program in Figure 2. All values are collected at the scheduler level.

In order to illustrate the other information provided by the benchmark, we use the EKF2 program implementing a Kalman filter. In Figure 3 we provide the ordered sequence of two input variables (the z coordinate of the gyroscope and the y coordinate of the accelerometer), showing different functioning modes for these two input variables.

In order to allow users to analyze such modes, we also provide the automatically detected transitions and subsequent clustering of the execution times of this program in Figure 4, where we used the procedure described in [14]. Any measurement-based WCET estimator may be used [6] and we use the Extreme Value Theory estimator [15], analyzing the statistical properties of these execution times. In Figure 5 we provide the estimation for all clusters according to two possible methods, the GEV, respectively, and the GPD estimators [15]. Two EVT estimators, GEV and GPD, first select maxima from an ordered sequence of execution times, and then they estimate the WCET distribution. The GEV estimator uses the block maxima approach by dividing data into several blocks with the same size and select the maximum of each block, while the GPD estimator selects the largest values above a given threshold.

V. FUTURE WORK: TOWARDS MULTICORE BENCHMARKS FOR MEASUREMENT-BASED WCET ESTIMATORS

Given the simplicity of the considered single core microcontroller, we believe that a multicore version of our benchmarks is important for our community. We migrate the autopilot programs to a Navio2 board[2] combined with a Raspberry Pi 3 B+ board including an ARM Cortex-A53 multicore. We prepare the same level of information as described in Section IV. Moreover, interested users may migrate them to any available Linux/POSIX-based board. We prepare the same level of information as described in Section IV. In order to finalize this new set of benchmarks we identify two main difficulties: (i) the OS migration from the single core ARM Cortex-M4 included in the Pixhawk board and (ii) the evolution of the data protocol uORB for the new environment.

REFERENCES

[1] F. Nemer, H. Cassé, P. Sainrat, J. P. Bahsoun, and M. D. Michiel. Papabench: a free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, volume 4 of *OASICS*, 2006.

[2] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15 of *OASICS*, pages 136–146, 2010.

[3] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 55 of *OASICS*, pages 2:1–2:10, 2016.

[4] C. Maxim, A. Gogonel, I. Asavoae, M. Asavoae, and L. Cucu-Grosjean. Reproducibility and representativity: mandatory properties for the compositionality of measurement-based WCET estimation approaches. *SIGBED Rev.*, 14(3):24–31, 2017.

[2]https://navio2.emlid.com

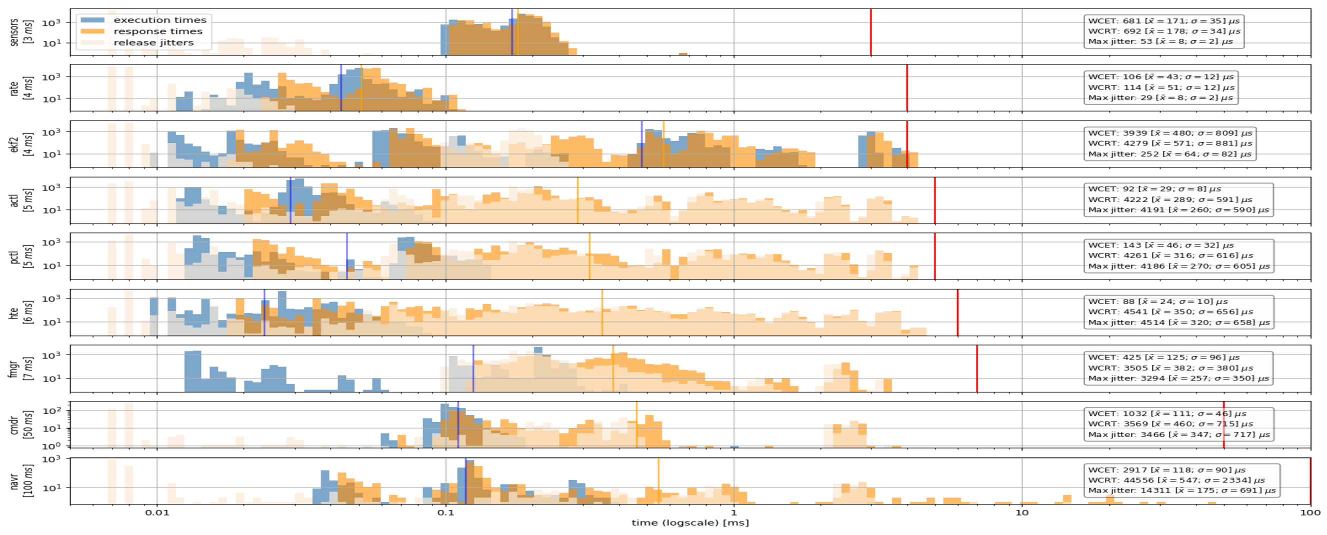

Fig. 2. Sequences of response times, execution times and release jitters

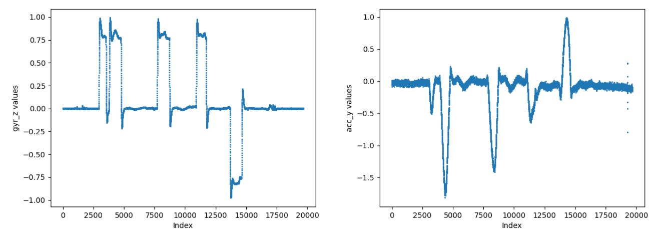

Fig. 3. Ordered sequence of EKF2 input variables: gyroscope (left) and accelerometer (right), one axis each

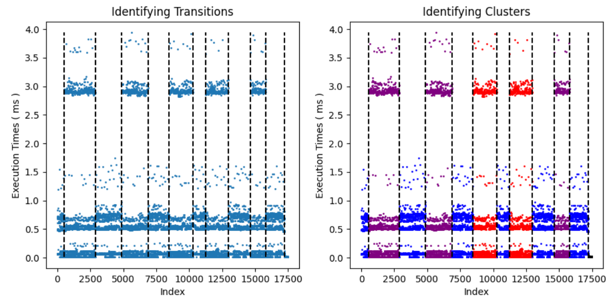

Fig. 4. Ordered sequence of EKF2 execution times with different transitions (left), three clusters color-identified (right)

[5] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study: From simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 309–318. IEEE Computer Society, 2014.

[6] R. I. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES*, 6(1):03:1–03:60, 2019.

[7] L. Meier. *PX4 Development Guide*. https://dev.px4.io/en/.

[8] G. Nuttx. *NuttX Operating System, User's Manual*. http://www.nuttx.org/doku.php?id=documentation:userguide.

[9] *MAVLink Developer Guide*. https://mavlink.io/en.

[10] *Gazebo Simulator*. http://gazebosim.org.

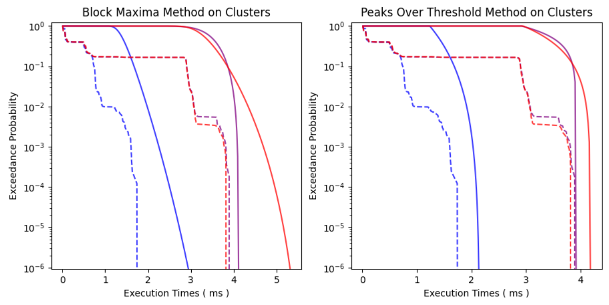

Fig. 5. WCET estimation for all clusters of EKF2 execution times, with GEV and GPD estimators, respectively. Colors correspond to those in Figure 4

[11] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[12] K. Zagalo, L. Cucu-Grosjean, and A. Bar-Hen. Identification of execution modes for real-time systems using cluster analysis. In *25th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2020*, pages 1173–1176. IEEE, 2020.

[13] S. Ben-Amor, D. Maxim, and L. Cucu-Grosjean. Schedulability analysis of dependent probabilistic real-time tasks. In *the 24th International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

[14] M. Wehaiba el Khazen, L. Cucu-Grosjean, A. Gogonel, H. Clarke, and Y. Sorel. Work-in-progress abstract: Wks, a local unsupervised statistical algorithm for the detection of transitions in timing analysis. In *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, pages 201–203. IEEE, 2021.

[15] M. Wehaiba El Khazen, A. Gogonel, and L. Cucu-Grosjean. Work in Progress: Lessons learnt from creating Extreme Value Libraries in Python. In *the 41st IEEE Real Time Systems Symposium (RTSS)*, December 2020.