An Application Platform for Embedded Real-Time Systems

Jun Wu*, Chin-Shung Hwang, Jun-Yu Liau, Jian-Liuang Huang and Ming-In Kuo

Abstract—**This paper presents an ongoing project to build an application development and execution platform for Linux-based embedded systems called Yapers. To avoid the peripheral failure problem from traditional super-loop architecture, Yapers supports real-time task and event-driven models such that an application consists of periodic real-time tasks and tasks triggered by certain events. The primary objectives of Yapers are: (1) to help developers build applications for embedded systems more efficiently and (2) to schedule and execute real-time tasks without violating their timing constraints. To accomplish the former objective, periodic and aperiodic task templates are provided to implement tasks more efficiently. Furthermore, Yapers provides a pool of shared variables, files, and database objects such that various forms of data can be shared/ exchanged between different tasks. An API library, called YapersLIB, is also provided to access shared data for ease of use. Developers only need to provide (1) tasks that comprise the application, and (2) configuration files that define the parameters and options for the system; the application will be generated by synthesizing an image with a patched operating system, booting scripts, tasks, required libraries and tools. This paper also provides an example of how to use Yapers to develop real-world applications.**

Index Terms—**Embedded real-time systems, embedded applications, application platforms, real-time tasks, aperiodic tasks.**

I. INTRODUCTION

In recent decades, embedded systems have been widely used in many domains, such as medical, transportation, aerospace, automotive, industrial automation, security, surveillance, etc. Accordingly, the need for varied embedded applications has increased. However, most embedded systems are resource-constrained compared to desktop PCs and servers [1, 2]. It is difficult to develop applications for a system with limited computing capabilities, less memory, and power. Furthermore, the hardware diversity of embedded systems also increases the difficulty even further. It decreases the reusability of program code and the portability of applications. In order to overcome those difficulties, many approaches have been proposed (such as [3–10]) for embedded systems. However, more work must be done for embedded real-time systems [11], which must guarantee that all the task instances can be completed without violating their strict timing constraints.

Manuscript received September 7, 2023; revised November 12, 2023; accepted December 29, 2023.

This work was supported in part by the National Science and Technology Council of Taiwan under a grant 110-2221-E-153-001-MY3.

J. Wu, C. S. Hwang, J. Y. Liau, J. L. Hwang and M. I. Kuo are with the Department of Computer Science and Information Engineering, National Pingtung University, 900 Pingtung City, Pingtung County, Taiwan

*Correspondence: junwu@mail.nptu.edu.tw

In this paper, an ongoing project, called Yet another platform for embedded real-time systems (Yapers), is presented to support real-time task model for Linux-based embedded systems. The primary goal of Yapers is to help developers build applications for embedded systems more efficiently. Since Linux has become one of the most popular operating systems for embedded systems. Yapers supports Linux-based embedded systems for which the hardware diversity issue has been reduced predominantly. As a result, an application developed by Yapers, denoted YaperAPP, can be executed on many Linux-based embedded systems with/without minor modification. Under Yapers, developers only need to provide (1) tasks that comprise the application, and (2) configuration files that define the parameters and options for the system, a YapersAPP will be generated and synthesized as an image with a patched Linux operating system, booting scripts, tasks, required libraries and tools. At the run-time, every running YapersAPP has a pool of shared variables, files, and database objects, which can be accessed by YapersLIB's APIs to communicate and share data between tasks. Another subproject, called YapersServer, provides the capabilities to manage multiple YapersAPP's and to access shared data from different YapersAPP's.

Unlike many other development tools and solutions, Yapers doesn't use the well-known *super-loop architecture* as the application paradigm. The super-loop architecture uses an infinite loop to perform a sequence of operations such that the functionalities of an application can be implemented. However, a potential issue is that the entire system might stop running when an operation fails in the super loop (due to an execution or a peripheral failure). To overcome the issue, Yapers uses the real-time task model and event-driven task model. We consider an application consisting of a set of periodic/aperiodic real-time tasks and tasks triggered by specific events. Using these two task models, an execution or peripheral failure at the run-time will not cause a YapersAPP to stop its execution. Furthermore, we have implemented a real-time task scheduler for Yapers to schedule tasks without violating their timing constraints (i.e., without missing their deadlines). Currently, two well-known real-time task scheduling algorithms (i.e., the rate monotonic (RM) [12] and the earliest deadline first (EDF) [12] scheduling algorithms) have been supported by the scheduler.

In the rest of this paper, Section II presents an overview of the Yapers project, which includes the software architecture, supported hardware and devices, and the current status of the project. Section III illustrates how to use Yapers to develop a real-world application for Raspberry Pi. Finally, Section IV is the conclusion and the future work of this research.

doi: 10.18178/jacn.2023.11.2.287

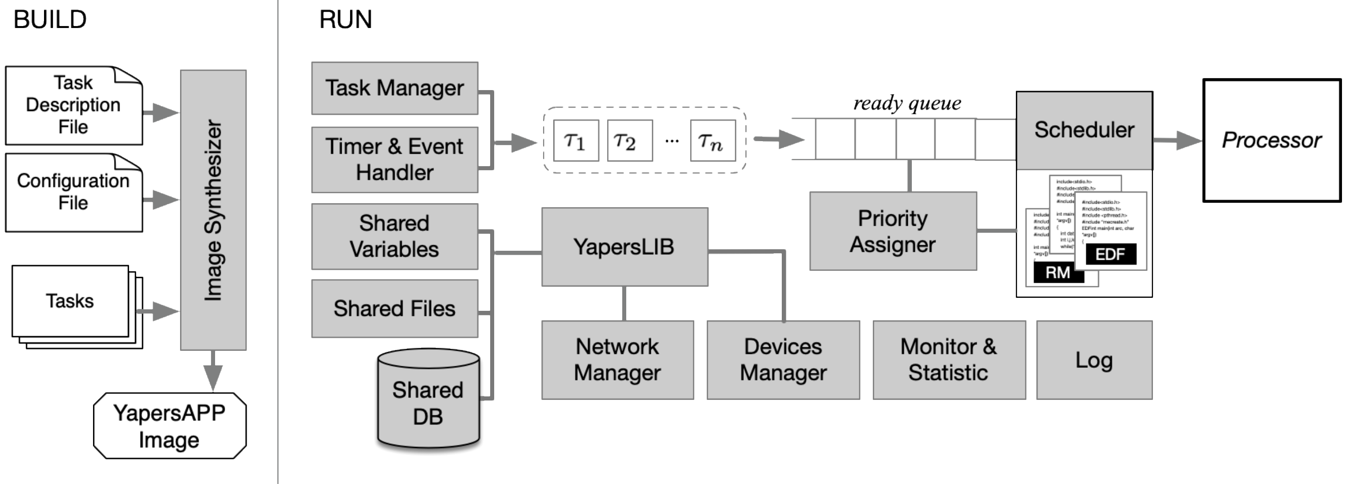

Fig. 1. Overview of Yapers.

II. YAPERS: AN EMBEDDED APPLICATION PLATFORM

In this section, an embedded application platform, called Yet another platform for embedded real-time systems (Yapers), is presented for the development and execution of embedded applications. Yapers is an ongoing project formerly known as meCreate [13, 14] designed originally for an energy-critical multiple versions task model. Yapers is not only a refinement, it further provides missing features to the project, such as mechanisms for data sharing between tasks, the API library for easy manipulate"on of peripheral devices and sensors, and the capabilities to manage multiple applications and target boards at the run-time. In the rest of this section, an overview, supported task models and target boards, the data sharing mechanisms, etc., shall be presented.

A. Overview

Yapers is an ongoing project that aims to build an application platform for Linux-based embedded real-time systems. At the current stage, we have implemented the platform prototype and built several real-world applications (i.e., YapersAPP). Fig. 1 shows the build stage and the run-time stage of a YapersAPP. In order to build an application upon Yapers, developers only need to provide a configuration file, a task description file, and a set of tasks at the build stage. The configuration file defines the target board's options and settings, including the network setting, task scheduling algorithm, etc. The task description file defines the parameters of tasks, such as arrival time, periodic, worst-case computation time, deadlines, trigger events, and minimum separation time. The *Image Synthesizer* is responsible for synthesizing an image that comprises the task set, booting scripts, and a patched Linux operating system. After the image has been synthesized, it can be downloaded into a target board to execute the application.

At the run-time, the *Task Manager* and *Timer & Event Handler* will generate task instances according to the parameters defined in the task description file (such as tasks' arrival time, periodic, trigger event, and minimum separation time). All the generated task instances will be put into the ready queue and scheduled by *Scheduler*. The *Scheduler* uses a dynamic priority scheduling algorithm (determined in the configuration file) to assign a proper priority to every task instance via *Priority Assigner*. As a result, all task instances in the ready queue can be appropriately scheduled. Note that Yapers also supports fixed priority scheduling algorithms for which the priority of each task's instances can be assigned statically. Furthermore, a task instance can share data with other tasks' instances via a pool of shared variables, files, and a shared database. Developers can use APIs provided in YapersLIB to access the shared data. To further analyze the workload and the usage, tools for monitoring data, usage statistics, and logs are also provided in Yapers.

B. Task Models

Traditionally, many embedded applications are implemented by using the well-known *super-loop* architecture—an infinite loop consisting of a sequence of operations. However, the super-loop architecture has a potential issue that the entire system might stop running because of a malfunction or a peripheral failure. In order to overcome the potential issue, Yapers use the *periodic real-time task model* [12] to treat each operation as a periodic real-time task. By using this task model, only the failure task will stop running while the other tasks are still running when a failure occurs.

Currently, Yapers supports two task models: *periodic real-time task model* and *event-driven task model*. We consider an application (i.e., YapersAPP) consists of a set of N periodic tasks $\mathcal{T}^{rt} = \{\tau_1^{rt}, \tau_2^{rt}, \cdots, \tau_N^{rt}\}$ and a set of M aperiodic event-driven task $\mathcal{T}^{ed} = \{\tau_1^{ed}, \tau_2^{ed}, \cdots, \tau_M^{ed}\}$. We describe these two task models as follows:

1) Periodic Real-Time Task: Each task $\tau_i^{rt} \in \mathcal{T}^{rt}$ can be defined by (A_i, T_i, C_i, D_i) where A_i, T_i, C_i, *and* D_i are the arrival time, period, worst-case computation time, and relative deadline, respectively. A periodic real-time task τ_i^{rt} will instantiate an instance $\tau_{i,1}^{rt}$ at its arrival time A_i, and then instantiate a new instance $\tau_{i,j}^{rt}$ (for $j \geq 1$) for every period of time (i.e.,T_i). Every task instance $\tau_{i,j}^{rt}$ has to complete its execution no later than the deadline (i.e.,$A_i + T_i \times j + D_i$).
2) Event-Driven Task: Each task $\tau_i^{ed} \in \mathcal{T}^{ed}$ can be defined by(EV_i, ST_i, C_i, D_i), whereEV_i, ST_i, C_i, and D_i are the triggered event, minimum separation time, worst-case computation time, and relative deadline, respectively. An event-driven task τ_i^{ed} will instantiate an instance as long as the event EV_i occurs. Let $\tau_{i,j}^{ed}$ denote the *j*th instance of τ_i^{ed} , and its instantiated time is defined

as$IT_{i,j}$. Every task instance $\tau_{i,j}^{ed}$ also has to complete its execution no later than its deadline (i.e.,$IT_{i,j} + D_i$). Note that for any two consecutive task instances $\tau_{i,j}^{ed}$ and $\tau_{i,j+1}^{ed}$ of an event-driven task τ_i^{ed}, we assume that$\left(IT_{i,j+1} - IT_{i,j}\right) \geq ST_i$.

C. Target Boards and the OS

Currently, Yapers supports Raspberry Pi 4 family and Linux kernel 5.10.17. We will expand it to support more target boards working with the Linux operating system. In particular, we are working on porting it to ASUS Tinker boards.

D. Task Scheduling Algorithms

Since Yapers supports two types of task models, we transform one of them to another for ease of scheduling. In particular, we create a fake real-time task τ_{N+i}^{rt} for each event-dirven task τ_i^{ed}, where $A_{N+i} = 0$, $T_{N+i} = ST_i$, $C_{N+i} = C_i$, and$D_{N+i} = D_i$. Note that the arrival time of the fake task is set as 0 because we can't predict the instantiated time of$\tau_{i,1}^{ed}$, and 0 is the worst-case consideration. After the transformation, a YapersAPP is considered to have a set of N+M real-time tasks $\{\tau_1^{rt}, \tau_2^{rt}, \cdots, \tau_N^{rt}, \tau_{N+1}^{rt}, \tau_{N+2}^{rt}, \cdots, \tau_{N+M}^{rt}\}$. These N+M tasks are scheduled by the *Scheduler* as shown in Fig. 1. We have implemented two well-known real-time task scheduling algorithms: Rate Monotonic (RM) [12] and Earliest Deadline First (EDF) [12]. Developers can decide on the scheduling algorithm in the task description file.

E. Shared Data

When the traditional super-loop architecture is adopted, operations in the loop can share data simply by using variables. However, a YapersAPP consists of a set of real-time tasks and event-driven tasks. For many applications, there is a need to share data between different tasks. Therefore, we have designed a mechanism in Yapers to handle the need. In particular, Yapers provides a pool of shared variables, files, and database objects such that various forms of data can be shared/exchanged between different tasks. Currently, Yapers only supports 32-bit int, 32-bit float, and 8-bit char data types. By default, a running YapersAPP has 20 shared variables identified by VID 0 to 19. We also provide 20 shared files and 10 shared database objects that are identified by FID and DBID. Note that we also use sv_i, sf_i, sdb_i to denote the shared variable, shared file, and shared database object with ID *i*, respectively. When an FID is specified, developers can read/write data from/to the corresponding shared files. Each database object represents a document of a NoSQL database (i.e., MongoDB). Note that the number of shared variables, files, and database objects can be defined in the task description file.

F. YapersLIB

To ease the development of YapersAPP, we provide an API library, called YapersLIB, to help developers access shared data (including shared variables, files, and databases) and to manipulate peripheral devices. In fact, a task of a YapersAPP is a program in Linux (and a task instance is a process). Yapers can support programs written in all languages. However, we only provide Python at the current stage since Python is getting more and more popular in the field of embedded systems. The following Python functions are selected from YapersLIB, which provide the capabilities for accessing share data:

```
yapersLIB.get_sv(vid:int) -> Any
yapersLIB.set_sv(vid:int, value:Any)

yapersLIB.openfile(
    fid: int,
    mode: OpenTextMode | OpenBinaryMode
) -> FileIO | TextIOWrapper

yapersLIB.copyfile(
    sourcefile: int | str,
    destfile: int | str
)

yapersLIB.insert_sdb(dbid:int, key:str, data:Any)
yapersLIB.remove_sdb(dbid:int, key:str)
yapersLIB.update_sdb(dbid:int, key:str, data:Any)
yapersLIB.find_sdb(dbid:int, key:str) -> Any
yapersLIB.empty_sdb(dbid:int)
```

III. A REAL-WORLD YAPERSAPP

Using Yapers, applications are easy to build for Linux-based embedded real-time systems. This section provides a real-world example to demonstrate the procedures for building a YapersAPP. The application is a campus surveillance system placed in National Pingtung University, called Safe@NPTU, which aims to improve security around our campus. The Safe@NPTU is deployed on Raspberry Pi 4, and it is equipped with a camera, speakers, flame sensor, temperature and humidity sensor, PM 2.5 sensor, sunlight sensor, gas sensor, and six 18600 Lithium-Ion battery (which provides up to 13,200 mAh power bank), as shown in Fig. 2. With the helps from Yapers, developers only need to provide a configuration file, a task description file, and a set of tasks, a YapersAPP's image will be synthesized. For Safe@NPTU, we have designed and implemented a set of real-time tasks and event-driven tasks for detecting and monitoring campus security. Table I shows some selected tasks (and their parameters) from Safe@NPTU.

As shown in Table I, task τ_{11}^{rt} gets the value from the flame sensor, and saves it in shared variable sv_0 for every 3000*ms*. τ_2^{ed} is an event-driven task, and it will be triggered when the value of the shared variable sv_0 equals to 1 (i.e., the flame has been detected by τ_{11}^{rt}). τ_2^{ed} will sound the speaker for 1 second, then the value of sv_0 will be reset to 0. Similar to τ_{11}^{rt}, τ_{15}^{rt} gets the value of gas concentration (value ranged from 0 to 1024) from the gas sensor, and saves it in shared variable sv_1 for every 3000*ms*. Another task τ_{16}^{rt} will check the value of sv_1 for every 3000*ms*. Since the value of sv_0 will be set to 1 if the gas concentration is getting higher (i.e., higher than 200), τ_2^{ed}will be triggered to sound the speaker. Task τ_{21}^{rt} and τ_{22}^{rt} act in a similar way to take a photo via the camera for every 5000*ms*, and then it will check the differences between any two consecutive photos (which are saved in shared files sf_0 and sf_1 alternately). According to the check result, it will sound the speaker when abnormal behavior is detected (i.e., two consecutive photos taken in a closed space have significant differences). Note that Yapers will monitor the status, values, or contents of all the shared data (including

shared variables, files, and database objects). Once a shared data has changed or meets some condition, Yapers will fire an event to notify related tasks. For example, τ_2^{ed} is an event-driven task, and it will be triggered when the condition $sv_0 == 1$ is met.

TABLE I: SELECTED TASKS OF SAFE@NPTU.

Task ID	Task File Name	Type	Triggering Event	Arrival Time	Period/Minimum Separation Time	WCET	Deadline	Description
τ_{11}^{rt}	flame_detect.py	P		0	3000	170	3000	Check the status of the flame sensor and save the value to sv_0.
τ_{15}^{rt}	gas_detect.py	P		0	3000	210	3000	Get the gas concentration (0-1024) from the gas sensor and save it to sv_1.
τ_{16}^{rt}	check_gas.py	P		0	3000	150	3000	Set $sv_0 = 1$ if $sv_1 > 200$.
τ_{21}^{rt}	take_photo.py	P		0	5000	1610	5000	Take a photo from the camera and save it to sf_0 and sf_1 alternately.
τ_{22}^{rt}	check_photo.py	P		0	5000	150	5000	Check the differences between sf_0 and sf_1. Set $sv_0 = 1$ if the differences are high.
τ_2^{ed}	alert.py	ED	$sv_0 == 1$		1160	150	1160	Sound the speaker for 1 second and reset sv_0.

* Type P and ED stand for periodic real-time tasks and event-driven tasks.
** The unit of time in this table is millisecond (*ms*).

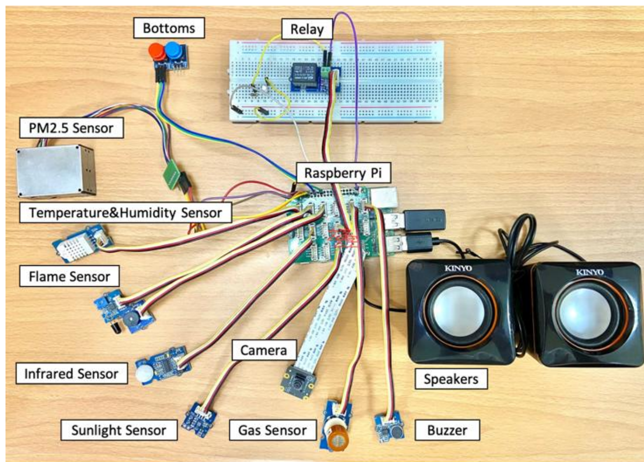

Fig. 2. An example YapersAPP — Safe@NPTU.

IV. CONCLUSION AND FUTURE WORK

This paper aims to provide an application platform, called Yapers, for building and executing applications on embedded real-time systems. At the current stage, the implementation of Yapers' prototype has been completed. Several real-world applications also have been built upon Yapers such that the correctness and efficiency are verified. In future work, we shall work on several subprojects, including YaperSERVER and YapersDEV. YapersSERVER is a server designed for managing multiple YapersAPP's, allowing a task to access the shared data from different YapersAPP's. YapersDEV is an integrated development environment for designing, building, and synthesizing YapersAPP's. We are also working on refining Linux so that Yapers can work with a more reliable, efficient, and predictable operating system.

REFERENCES

[1] P. Koopman, "Embedded system design issues (the rest of the story)," in *Proc. the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, Austin, TX, USA, 1996, pp. 310–317.

[2] T. A. Henziger and J. Sifakis, "The embedded systems design challenge," in *Proc. of the 14th International Symposium on Formal Methods, Hamilton*, Canada, August 21-27 2006, pp. 1–15.

[3] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented designof embedded hardware and software systems," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 231–260, 2003.

[4] C. F. Hsu, "A component-based software development platform for rapid prototyping of embedded software," Master Thesis, Department of Computer Science and Information Engineering, National Pingtung University, July 2009.

[5] J. Wiklander, J. Eliasson, A. Kruglyak, P. Lindgren, and J. Nordlander, "Enabling component-based design for embedded real-time software," *Journal of Computers*, vol. 4, no. 12, pp. 1309–1321, 2009.

[6] J. Wiklander, J. Eriksson, and P. Lindgren, "An IDE for componentbased design of embedded real-time software," in *Proc. of 6th IEEE Int'l. Symposium on Industrial & Embedded Systems*, 2011, pp. 47–50.

[7] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," in *Proc. of the IEEE*, vol. 91, no. 1, 2003, pp. 145–164.

[8] N. Hili, C. Fabre, S. Dupuy-Chessa, and D. Rieu, "A model-driven approach for embedded system prototyping and design," in *Proc. IEEE International Symposium on Rapid System Prototyping*, New Delhi, India, 2014, pp. 23–29.

[9] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[10] J. Fu, Y. Jiang, W. Ren, and D. He, "A hardware and software programmable platform for industrial embedded application," in *Proc. 2015 Chinese Automation Congress (CAC)*, 2015, pp. 360–365.

[11] K. V. Prashanth, P. S. Akram, and T. A. Reddy, "Real-time issues in embedded system design," in *Proc. the 2015 International Conference on Signal Processing and Communication Engineering Systems*, 2015, pp. 167–171.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, 1973.

[13] J. Wu and J. L. Wang, "A real-time embedded platform for mixed energy-criticality systems," in *Proc. of the 7th IEEE ICASI*, 2021.

[14] J. Wu and J. L Wang, "An energy-efficient embedded system platform for energy-critical real-time tasks," *Engineering Letters*, vol. 31, no. 1, pp. 105-112, 2023.

Copyright © 2023 by the authors. This is an open access article distributed under the Creative Commons Attribution License (CC BY-NC-ND 4.0), which permits use, distribution and reproduction in any medium, provided that the article is properly cited, the use is non-commercial and no modifications or adaptations are made.