Real-Time Operating Systems for Cyber-Physical Systems: Current Status and Future Research

Anthony Serino *
Computer Science Program
Misericordia University
Dallas, PA 18612
serinoa1@misericordia.edu

Liang Cheng
Department of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015
cheng@lehigh.edu

Abstract—**This paper studies the current status and future directions of RTOS (Real-Time Operating Systems) for time-sensitive CPS (Cyber-Physical Systems). GPOS (General Purpose Operating Systems) existed before RTOS but did not meet performance requirements for time sensitive CPS. Many GPOS have put forward adaptations to meet the requirements of real-time performance, and this paper compares RTOS and GPOS and shows their pros and cons for CPS applications. Furthermore, comparisons among select RTOS such as VxWorks, RTLinux, and FreeRTOS have been conducted in terms of scheduling, kernel, and priority inversion. Various tools for WCET (Worst-Case Execution Time) estimation are discussed. This paper also presents a CPS use case of RTOS, i.e. JetOS for avionics, and future advancements in RTOS such as multi-core RTOS, new RTOS architecture and RTOS security for CPS.**

Index Terms—**Real-time Operating Systems; Worst-Case Execution Time; Cyber-Physical Systems.**

I. Introduction

Real-time operating systems (RTOS) are used in a wide range of cyber-physical systems (CPS) including industrial systems, such as process control systems, avionics, and nuclear power plants. Most RTOS run on embedded systems consisting of pieces of hardware that work as controllers with dedicated functions within mechanical and/or electronic systems. Real-time operating systems are critical for those mechanical and/or electronic systems with real-time requirements because they could not be operated safely without RTOS. In many real-time CPS a missed deadline can lead to disastrous consequences.

A. *Hard Real Time vs. Soft Real Time Systems*

Cyber-physical systems include hard real-time systems and soft real-time systems [12]. The primary difference between hard real-time and soft real-time is that the consequences of missing a deadline differ from each other. For instance, performance (e.g. stability) of a hard real-time system such as an avionic control system or a nuclear power plant, is dependent on both the timeliness of the operation results and the correctness of the results. However, for soft real-time systems such as a multimedia on-demand system, their performance is largely dependent on the results. A hard real-time system is used in systems that are time sensitive and must meet their deadlines in order to avoid system failures. However, deadlines in soft real-time systems are less strict so that if they miss their deadlines it does not result in disasters. Note that hard real-time systems do not have an easy way to recover from a failure, whereas a soft real time system can be time-elastic.

* This work was done when A. Serino was with a REU program hosted by the Department of Computer Science and Engineering at Lehigh University.

B. *Challenges and Contributions*

When researchers and engineers design and implement time-sensitive CPS, it is important and challenging to select and optimize RTOS suitable for the targeted applications. There are many existing studies on RTOS and their results are scattered in a large number of publications. Therefore, CPS researchers and engineers may spend a lot of time in sifting through papers and comparing various RTOS techniques to select a proper RTOS for their applications while considering many factors such as scheduling mechanisms and priority inversion solutions influencing their selection decisions.

This paper surveys CPS-related RTOS technologies, discusses select topics including new adaptations on solving priority inversion, a newly purposed platform for WCET (worst-case execution time) tool development, and a RTOS that supports multi-core processing. The contribution of this paper is that it provides a quick reference of RTOS technologies for CPS researchers and engineers and helps them identify the proper RTOS and techniques for their CPS applications.

This paper starts with a comparison between GPOS and RTOS and discusses their advantages and disadvantages for general computing and for real-time cyber-physical systems. Section II presents different techniques used to handle task scheduling and address priority inversion. Section III provides a comparison between three RTOS implementations, namely VxWorks, RTLinux, and FreeRTOS. This comparison covers kernels of the operating systems, schedulers used, and how they solve priority inversion. The next section discusses WCET analysis tools used today in industry along with the development of a new platform to build and compare these tools. Section V provides a use case of a recently developed RTOS, i.e. JetOS in avionics. The final section of this paper discusses future research and developments of RTOS for CPS ranging from a multi-core RTOS (HIPPEROS) to a new RTOS platform (HERCULES).

TABLE I: Priority Inversion Methods

Priority Inversion Method	Ensures execution of the highest priority task	Swaps the priority of tasks	Expands the amount of priorities available	Finishes running lower priority tasks to free up critical sections
Priority Inheritance	✓			✓
Priority Ceiling	✓			✓
Priority Remapping	✓		✓	✓
Priority Exchange	✓	✓		✓

II. GPOS vs RTOS

A general comparison between RTOS and GPOS is provided in this section to show the advantages and weaknesses of these operating systems when used for real-time cyber-physical system applications.

A. *Task Scheduling*

The first part of differences is the way where GPOS and RTOS perform their task scheduling. Generally speaking, a GPOS utilizes a fairness policy that allows all processes to share the processor. This hinders the GPOS' ability to handle time sensitive tasks because it cannot guarantee task dispatch latencies. In contrast, RTOS use priority based preemptive scheduling mechanisms to enable high priority tasks to take the processor from lower priority tasks and allow the high priority tasks to run without interruption. A RTOS may also utilize every resource at its disposal to get peak performance.

1) Scheduling techniques:

a) Round robin scheduling: This form of scheduling uses processor time sharing and gives every process with the same priority a set of time slices [4] [5], each of which corresponds to a fixed amount of processor or CPU time. When a process uses up a CPU time slice the scheduler forces it out of the CPU so that it takes turns to use the CPU resource with other processes. This may lead to relatively large overhead of context switching.

b) First-in-first-out (FIFO) scheduling: The FIFO scheduler will try to execute the task with the highest priority first, but when the processes have the same priority they are scheduled in the order of their arrivals. The first task there will run to completion before starting the next one, or until a higher priority task takes the processor [4] [5].

c) Rate-monotonic scheduling: This scheduling method is a static-priority algorithm that sets the priority level for each task in the order of their period information. Short period tasks execute frequently, and a long period refers to infrequent execution. Short period tasks are given higher priorities while long period ones are given low priorities. This lets high priority tasks run first, and it is best used when there are well defined periodic tasks with the same CPU run time length [4] [5].

d) Earliest-deadline-first (EDF) scheduling: This scheduling method computes the priority of processes dynamically based on their arrival time, execution requirements and deadlines so that it schedules the task with the earliest deadline first [4] [5]. EDF scheduler is more capable of making all deadlines to be met when system load is high comparing to rate-monotonic scheduling.

2) Kernel: Generally the GPOS does not support preemption. Preemption is when processes and threads with higher priorities can take the processor from lower priority tasks. By not allowing preemption the GPOS suffers when trying to complete a task that is time sensitive as it can make the task wait and thus miss its deadline. The GPOS is unable to cancel system calls even if they are from a lower priority task, which leads to unpredictable delays. The advantages of its kernel are in its support for widely used application programming interfaces (APIs) and customizable operating system components for application-specific demands.

The RTOS fully supports preemption where it imposes an upper bound on how long the preemption has its interrupts disabled. This allows a high priority task to run to completion without interruption and thus let time sensitive tasks in CPS meet their deadlines. The kernel of a RTOS tries to use the least amount of resources possible. To keep it simple only services with short execution paths are allowed in the kernel, and its process loading happens outside along with its file systems. This architecture makes it so that if one of these systems fails it does not corrupt other services or the kernel. The benefit of the RTOS kernel is that there is only a small core of fundamental operating system services in the kernel, which are signals, timers, and the scheduler [14].

B. *Priority Inversion*

Priority inversion occurs when a high-priority task (*H*) shares a resource or critical section with a low-priority task (*L*) and waits for the *L* to finish its task while *L* is preempted by a middle-priority task (*M*), which does not use the shared resource nor run the critical section. In this scenario *H* has to wait for *M* to finish so that the control can be given back to *L* for its release of the shared resource or completion of the critical section, and thus in effect the higher priority task has to wait for the middle-priority task even though *H* does not shared any resource or critical section with *M*. This is an issue for time sensitive tasks as it can prevent them from meeting their deadlines by forcing them to wait for the availability of needed resources. This needs to be fixed as it can result in blocking including chain blocking or worse a full deadlock. There are a few protocols to solve priority inversion.

Table I lists and compares different priority inversion methods. It shows how these methods can manipulate the priority of tasks to achieve priority inversion inside RTOS. Most of these methods are expansions of *Priority Inheritance*, which let *L* inherit the priority of *H* when *L* is using the shared resource or in the critical section at the time when *H* starts pending for the shared resource of the critical section. Both *Priority*

Remapping and *Priority Exchange* use priority inheritance as a base for their priority inversion solutions. The key difference is the adjustment that these methods make to *Priority Inheritance*. *Priority Remapping* doubles the amount of priorities assigned to tasks, but only using the added odd ones as marks for lower priority tasks trying to attain resources when there is a higher priority task using the resources. *Priority Exchange* changes the way in which it gets the lower priority task to run such that it will not indefinitely restrict the critical section. *Priority Ceiling* is unique in the bunch listed here as it tests whether a task to be scheduled can reach a higher priority than that of the current running task and all its inherited properties, and if it fails to meet this priority then the pending task gets suspended.

1) Priority inheritance:

The priority inheritance protocol is where if a higher priority process is waiting for a lower priority process for a shared resource the process scheduling algorithm gives the higher or the highest priority to the lower priority task on the processor so that it cannot be preempted by a different task until it completes the execution of its critical section [2]. For example, there is a process A on the processor of priority 2 and there is a higher priority process B of priority 7 that could not preempt it. The process scheduling algorithm would assign process A priority 9 temporarily so that it finishes its critical section on the processor without being interrupted by other processes which could delay process B further. This allows process B to get access to the resource as fast as possible, and once the resource is freed by process A the process scheduling algorithm reverts process A's priority back to its original priority of 2.

2) Priority ceiling:

The priority ceiling is where each resource is assigned a priority ceiling where the priority is equal to the highest priority of any task which may lock the resource. This works by temporarily raising the priority of tasks in certain situations. Basically if process A tries to preempt the critical section of another process to execute in its own critical section, then the priority of the new section should be higher than the priority of inherited properties of all the preempted sections. If this fails then process A is denied entry into the critical section and is suspended [2].

3) Other priority inversion solutions:

a) Priority remapping:

As an improvement to the priority inheritance protocol, the priority remapping method expands the highest priority from 64 to 128 without changing the interface of the task creation function by multiplying the priority of a task by 2 as its internal priority. This means that users still only see 64 priorities while their internal priority set is extended to $\{0,2,4,...,126,128\}$. The odd internal priorities are left for changing when priority inversion happens.

b) Priority exchange:

This method is also an improvement of the priority inheritance method. It swaps the tasks' priorities when a higher priority task is blocked by a lower priority task. The priorities will be swapped back after the lower priority task finishes running the critical section [3].

C. Modified GPOS

A modified GPOS is the result of adapting and changing GPOS to have the same capabilities of both GPOS and RTOS. It has a major advantage in that if it works like GPOS it can be used more widely for a greater amount of tasks, and if it has the capabilities of RTOS then it supports tasks for time critical CPS. For example, Linux 2.6 added preemption [11]. Generally when a GPOS is directly modified to support RTOS functionality high-resolution timers will be used. This modification enables the process scheduler to make the system more reactive and event-driven. However, it is not as fast as other RTOS and the low latency patches for the GPOS timers do not solve the priority inversion issue [14].

Another way to improve GPOS is by introducing a new architecture called dueling kernels where the GPOS is ran on top of an RTOS [14]. This architecture sends real-time tasks to run on the RTOS, with a higher priority than other tasks running on the GPOS. The RTOS gives these tasks the ability to preempt the tasks on GPOS, and then gives the CPU back to the GPOS when it finishes running the high priority tasks. This system has an issue where the tasks running on the RTOS have limited use of the GPOS services due to preemption issues. This causes RTOS to recreate services that exist in the GPOS. RTOS tasks also cannot use the memory management unit which is used by the GPOS for non-realtime processes. GPOS services that are ported often have different vendor extensions that do not work with other vendor's extensions.

III. VxWorks, FreeRTOS, RTLinux

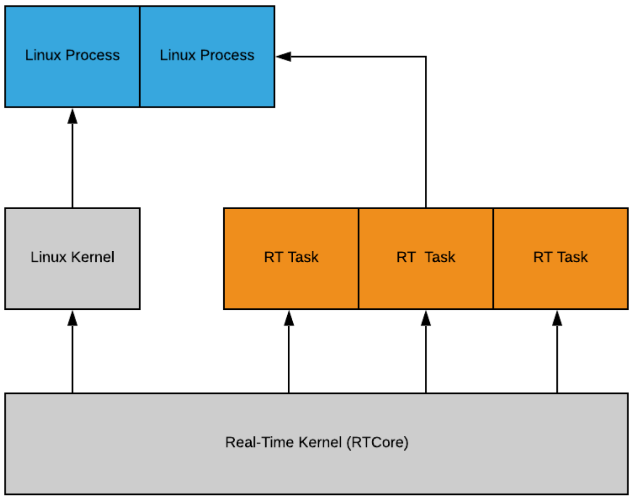

Figure 1. RTLinux Architecture Overview

The RTOS chosen for comparisons in this paper are some top competing RTOS in industry. VxWorks and RTLinux have been extensively compared to each other through research due to the continuing development on both the commercially available VxWorks and the free RTLinux. FreeRTOS being a more recent RTOS compared to VxWorks and RTLinux has seen little comparisons with either VxWorks or RTLinux directly. Important aspects to compare them include their kernels, schedulers, and how they handle priority inversion.

A. Kernel

1) RTLinux: RTLinux has a special design for its kernel because it has two kernels as illustrated in Figure 1. RTLinux uses a specialized real-time kernel called the RTCore [4]. The second kernel is the standard Linux kernel which is used for regular applications that do not have time constraints. Both interrupt handling and thread handling are controlled by RTCore, which will send these interrupts to the appropriate interrupt handler. The RTCore also restricts the Linux kernel by making it unable to disable interrupts to make sure it does not interfere with process scheduling. Thus the Linux kernel can only run when there is a task that is not realtime. Real-time applications can communicate with Linux kernels through first-in-first-out pipes. The duel kernels gives RTLinux the full functionality of Linux while adding real-time capabilities [6].

2) VxWorks: VxWorks uses a single micro-kernel to handle basic kernel functions [4]. Additional functions like file sharing and networking have to be loaded from provided libraries. This system provides flexibility to fit its functionality without loosening its constraints on available memory and resources [8] [9].

3) FreeRTOS: FreeRTOS also utilizes a single micro-kernel to handle real-time tasks. This kernel supports dynamic scheduling or a priority based scheduler, blocking and dead-lock avoidance, and scheduler suspension. It can utilize fully featured API, or a lightweight API [8] [9].

These kernels are similar in how they all have a means to handle real-time tasks. Some of the major contrasts for RTLinux is that it supports a duel kernel which allows it to handle a wide variety of tasks at the cost of being a larger kernel by having both the standard Linux kernel and the RTCore. The architecture of VxWorks and FreeRTOS are similar in using micro-kernels.

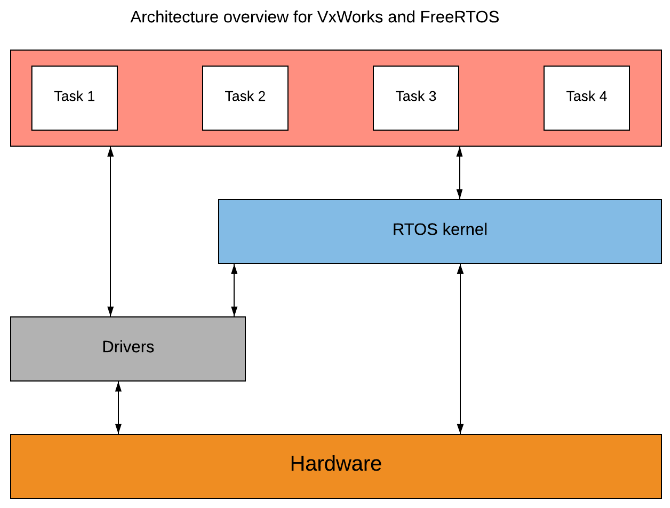

Figure 2. Architecture for VxWorks and FreeRTOS

B. Scheduler

The scheduler of RTOS is an important part of how an RTOS decides the next task to be run on the processor, and to make sure that all tasks meet their deadlines. This section will discuss similarities and differences between the schedulers used by RTLinux, VxWorks, and FreeRTOS.

1) RTLinux: RTLinux has a flexible scheduler by allowing different scheduling techniques to be used based on the program's needs. The RTLinux scheduler supports FIFO scheduling, EDF scheduling, and rate-monotonic scheduling. This enables different systems to use different schedulers suitable for achieving their real-time requirements [5].

2) VxWorks: VxWorks uses a preemptive round-robin scheduling algorithm. Task priorities can range from 0 to 255 where 0 is the highest priority [11]. If a task with a higher priority than the one on the processor is ready to run, then the lower priority task will be suspended so that the higher priority task can be ran. If the two tasks have the same priority then they go into round-robin scheduling. If a resource is unavailable then the processor swaps back to the lower priority task until the resource is available. VxWorks supports POSIX API which makes the system FIFO. This provides flexibility to meet different industrial needs [5].

3) FreeRTOS: FreeRTOS uses a dynamic preemptive priority based scheduling algorithm [9]. This scheduler can allow the user to choose running processes in a cooperative manner or using a preemptive policy. The difference is that the preemptive policy always runs the highest priority task, and when two tasks have the same priority they share CPU time. The cooperative manner allows context switches to occur by calling a function or when a task gets blocked.

Both RTLinux and VxWorks use the priority inheritance protocol. However, FreeRTOS does not use one of the typical ways of dealing with priority inversion; it deals with deadlocks formed by priority inversion by enforcing non-blocking tasks and by blocking tasks for fixed amounts of time.

IV. WCET TOOLS

As delays may impact the stability and correctness of cyber-physical systems, it is important to analyze the delays introduced by the operating systems [17] and by the networks [22] for CPS applications. WCET (Worst-Case Execution Time) analysis tools give an estimated worst case execution time of a task. Modern processor components like caches and pipelines complicate the task of estimating the WCET [17]. If a tool does not take pipeline or cache behaviors into account it may overestimate the WCET by multiple orders of magnitude.

A. AbsInt aiT

AbsInT aiT is a worst-case execution time analysis tool. It addresses the cache and pipeline issue by statically analyzing a task's cache and pipeline behaviors, which enables obtaining a correct upper bound of WCET of the task. This tool uses the technique of abstract interpretation, offers a graphical user interface to visualize the WCET path, and allows for an interactive way to inspect pipelines and caches [17]. The

TABLE II: WCET Tools

Tool Name	Analysis Target	Analysis Method	Goal	Source
AbsInt aiT	Cache and pipeline behaviors	Static analysis and abstract interpretation	Find WCET for RTOS	https://www.absint.com/ait/index.htm
Bound-T Tool	Code	Presburger-arithmetic based analysis	Find WCET for RTOS	http://www.bound-t.com/
OTAWA Toolbox	N/A	N/A	Help compare WCET tools to each other	https://www.tracesgroup.net/otawa/

abstract interpretation uses a semantic based method for safe and static program analyses.

The overall method applied by AbsInT aiT has multiple steps in finding the WCET. (1) Reconstruction of the control flow; (2) Value analysis: computation of address ranges for instructions' access memory; (3) Cache analysis: classifies memory references as cache hits or misses; (4) Pipeline analysis: predicts the behavior of the program on the processor; (5) Path analysis: determines the worst case execution path; and (6) Analysis of loops and recursive procedures. Steps 2, 3, and 4 are done with abstract interpretation and the path analysis is done using integer linear programming [17].

B. *Bound-T Tool*

The Bound-T tool is software used for static analysis of code to estimate its WCET and stack usage for embedded systems. This tool in its current state is not going through further development, and it is currently open source. The Bound-T tool uses the same static analysis approach as that used by AbsInT aiT. The Bound-T tool was developed for local and safe analysis of simple control flows. It uses a PA (Presburger Arithmetic)-based analysis where an approximate model of computer arithmetic is used, which assumes that integer variables never overflow or wrap around. However, this approximation leads to an issue of not being able to find a feasible execution path. In order for the Bound-T tool to start getting back on track to find WCET for embedded systems it either has to drop the PA-based analysis or find a new form of preliminary analysis to ensure that the PA analysis can be applied [15].

C. *OTAWA Toolbox*

The OTAWA toolbox is a WCET analysis tool framework that supports hosting researchers' WCET algorithms and includes an abstraction layer that separates the hardware analysis from the instruction set architecture. This framework should also allow the comparison between tools and support the development of new tools. The OTAWA toolbox was used in the MERASA project to help find the most optimal WCET analysis tool for a multicore processor running mixed-critical workloads [16].

Table II summarizes and compares the WCET tools discussed in terms of analysis targets, analysis methods and design goals. Two out of these three tools are designed to find WCET of a task in RTOS. The OTAWA toolbox is designed to help researchers to be able to run multiple WCET tools on the same architecture.

Both the AbsInt aiT tool and the Bound-T tool use static analysis to find the WCET. The difference is that Bound-T has run into trouble in that it uses PA-based analysis that may not be able to reliably find upper bounds. The AbsInt aiT tool is a popular tool for specific processors. The OTAWA framework should allow for newly developed WCET analysis tools to be compared with existing tools, and help further development of WCET tools by offering a standard framework and C++ library [16].

V. A CPS Use Case of RTOS

Avionics is a good example where real time cyber-physical systems are used. There has been recent development of a specific type of RTOS, i.e. JetOS, to fully meet the ARINC653 international standard for aircraft usage. JetOS originates from POK RTOS, an open source project, which partially meets the ARINC653 standards. The critical parts that have been reworked or added to meet the standard include POK's scheduler, network stack, memory manager, added separate memory, and a reduction of the kernel size for less errors. The system uses ordinary partitions which separates memory, and system partitions are used to utilize services outside of the ARINC653 standard. Both of these types of partitions from the kernel point of view are the same. Currently there are a working prototype of JetOS and its extensions [13] [23].

The kernel of JetOS dropped POK's AADL (Architecture Analysis and Design Language) configuration tools for XML based configuration files. Furthermore it dropped the SPARC platform in favor of building JetOS on top of a platform that other avionic systems use. The platform chosen to replace it was the x86 and powerPC.

The kernel is built to support multiple schedulers because different partitions can utilize different schedulers, and it is configured statically where the number of partitions, partition memory size, port, names, etc. cannot be changed [13]. Each partition may have one or more processes. Partitions are scheduled based on a round-robin algorithm. Intra-partition schedulers implement lock-wait-unlock and priority scheduling. Resources are pre-allocated to ensure reliability. The memory is pre-allocated to every partition. These partitions are scheduled differently than kernel modules where partitions are run in the user mode with time and space constraints [13].

VI. Future Directions

With the rising need for both faster computing speed and better resource usage, multi-core computing has existed in general-use computers for a number of years. A multicore RTOS as new architecture for embedded systems and CPS needs to aim at providing a major increase in both speed and resource usage including energy efficiency [10].

A. Multicore RTOS

A challenge in developing multicore RTOS is how to evaluate it according to industry standards. Uni-core systems meet the requirements of standards such as ARINC653 and AUTOSAR. Researchers have put forward the use of general purpose operating systems, such as Linux, in order to provide an environment where they could evaluate a multicore real-time system. Although this approach has the advantage of being able to reuse code, it runs into issues where Linux was not built to support hard real-time systems or to meet the constraints for safety-critical applications. HIPPEROS, a multicore RTOS project launched in 2010, is built from scratch so that it can implement hard real-time techniques with multicore design principles and scale with an increasing amount of cores [10].

1) Kernel: HIPPEROS' kernel is able to run on multiple different architectures and platforms with an arbitrary number of cores. It uses distributed asymmetric micro-kernel architecture that allows each core to execute a local part of the kernel. This enables a dedicated core to execute the kernel's system calls, scheduler, and resource handling and frees up parts of the kernel for parallel processing. The kernel is configurable in that a CPS developer or system designer can select the scheduling policy or resource allocation protocol at its build time. To manage hard real-time tasks it uses a process model that gives executable and timing information like deadline, period, and worst-case execution time.

a) Asymmetric kernel architecture: The problem with symmetric kernel architecture design is that the cores are executed with the same kernel code and protected data structures with fine-grained lock mechanisms. The asymmetric design allows for one core to fully dedicate itself to running the scheduler and dispatching the processes to other cores. For the HIPPEROS project the dedicated core is called the master core and is responsible for managing global resources, scheduler, system calls, and message passing to allow the kernel to be executed in parallel. It works as follows. Whenever there is a scheduling decision the master core must be woken up to notify the slave core (any core other than the master core) to perform a context switch. The slave cores can only perform context switch after receiving an inter-processor interrupt (IPI) from the master core. This system of master and slave cores does not require locking mechanisms, and it is expected to be able to handle up to 8 cores before overloading the master core without using clustering for enhancement [1] [10].

2) Scheduler: The scheduler's API is preemptive and priority based. When a task switches state (e.g. blocked to ready), a scheduler module is called and it decides if context switching must occur according to tasks' priorities. If a task misses its deadline, then a configurability policy enacts a range of responses that can terminate the process, ignore the event, or change the priority of the process [10].

B. HERCULES Framework

HERCULES is a project for the development of a high-performance real-time architecture for low-power embedded systems. Estimated features of the HERCULES project include a reduction in energy consumption, productivity improvement in programming and maintaining advanced computing systems, increased concurrency and parallelism in applications, and enhanced trust of embedded systems. The objective of the project is to introduce predictability into embedded high-performance computing. Use cases of HERCULES include avionic and automobile cyber-physical systems.

The avionic use case considers future airplanes that will have more complex needs in regards to image processing or computer vision, which may be used during landing, surveillance activities, and navigation. Moreover, the number of cameras on board an airplane is expected to increase to support possible direct video streams for the pilot and crew and possible automation based on meaningful data extraction from the video streams. Machine learning techniques can be applied and have been proven to perform image processing at the cost of increasing algorithmic complexity and computational requirements. A visual object tracking application based on Airbus high-speed machine learning techniques has been used to test the HERCULES framework with various programming models and GPU-based platforms [7].

The automobile use case is involved with autonomous driving for valet parking. The HERCULES framework was chosen to be tested for valet parking for three reasons. First it has subset functionalities required for self-driving cars, and testing is affordable within the time frame of the HERCULES project. Second, the project team has simulators and test environments to create the scenarios. Third, the agents drive at low speeds, which offers clear safety advantages. This use case shows four major areas where algorithms will be applied: perception (sensor data processing), data fusion (creates environmental model), decision (action and path planning), and localization (GPS and MAP data management) [7].

These use case studies show the flexibility of the HERCULES framework, which can be adapted to different industries.

C. Security Research related to RTOS

As RTOS has been used in devices and SCADA (Supervisory Control And Data Acquisition) systems for time-sensitive and mission-critical tasks, its security is an important area of research. Several security vulnerabilities of RTOS have been discussed in the literature, such as lack of authentication enforcement, inefficiency of encryption, code injection, exploiting shared memory, priority inversion, denial of service attacks, and inter-process communication attacks.

Adversaries may attack the devices or embedded systems through accessing the remote debugging tools of RTOS. For example, US-Cert Vulnerability Note VU362332 [18] stated that VxWorks debug service was enabled by default so that an attacker might be able to fully comprise the embedded systems and create severe damages to the physical processes controlled by the SCADA systems through remote memory dump and remote function calls. Thus security should be a top priority when designing, implementing, and configuring RTOS.

RTOS security may be enhanced by detecting attacks based on systems call timing and sequences [20], network activities, and power consumption patterns of embedded systems [22]. For example, a class of attacks, known as payload attacks and successfully launched by Stuxnet, modifies PLC control programs (i.e., the "payload" for PLC firmware which may be a RTOS) and causes damages to the physical system. In [20] the authors studied firmware-level detection of PLC payload attacks that alter the timing behavior of the payload. WCET analysis and monitoring have also been used to enhance RTOS security [21].

RTOS also needs to address the memory fragmentation issue to improve its reliability and avoid program stalls as field devices such as PLCs (Programmable Logic Controllers) using RTOS may continuously run in an extensive period of time in months and even years without rebooting [19].

VII. Conclusion

With the advancement of real-time CPS applications such as autonomous driving and industry control, RTOS becomes an important topic for CPS researchers and engineers. This paper surveys the current status of RTOS and provides a quick reference of RTOS technologies to help researchers and engineers identify proper RTOS and related techniques for their CPS applications. This paper has introduced how RTOS have been developed from GPOS, and how they have different advantages and disadvantages for general computing and dealing with real-time requirements. We have compared three RTOS implementations that are used in industry, namely VxWorks, RTLinux, and FreeRTOS, in terms of their kernels, schedulers, and how they handle priority inversion. The paper also describes three tools used in industry for WCET analysis. The discussions of JetOS and HERCULES framework demonstrate how existing RTOS can be adapted and changed to meet CPS requirements. This paper also presents future directions for research and development of RTOS from the perspectives of multi-core systems and security.

Acknowledgment

The authors would like to thank the supports provided by the U.S. National Science Foundation (NSF) under Award Numbers CNS-1757787 and CNS/CPS-1646458, and by the U.S. Department of Energy under Award Number DE-OE0000779. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors of the research.

References

[1] Juan Rivas, Joel Goossens, Xavier Poczekajlo, Antonio Paolillo. Implementation of memory centric scheduling for COTS multi-core real-time systems. In Proceedings of the 31st Euromicro Conference on Real-time Systems. 2019.

[2] Lui Sha, Ragunathan Rakumar, John Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185. September 1990.

[3] Silambarasan, Ramanatha Venkatesan. Handling of priority inversion problem in RTLinux using priority ceiling protocol. In Proceedings of the International Journal of Advanced Engineering Research and Science (IJAERS). Vol. 3, Issue 6. June 2016.

[4] Daniel Forsberg and Magnus Nilsson. Comparison between scheduling algorithms in RTLinux and VxWorks. Technical Report, Computer Science and Engineering at the University of Linköping, November 2006.

[5] Stefan Holmer, Osker Hermansson. A comparison between the scheduling algorithms used in RTLinux and in VxWorks - both from a theoretical and a contextual view, Technical Report, Computer Science and Engineering at the University of Linköping, 2006.

[6] Federico Reghenzani, Glueppe Massari, William Fornaclari. The real-time Linux kernel: A survey on Preempt RT. ACM Computing Surveys. Vol. 52, No. 1, Feb. 2019.

[7] Marko Bertogna. High-Performance Real-time Architectures for Low-Power Embedded Systems. H2020-EU.2.1.1. Project Website: https://cordis.europa.eu/project/rcn/199161/factsheet/en. 2016-2018.

[8] Ming-Yuan Zhu. Understanding FreeRTOS: A Requirement Analysis. CoreTek Systems, Inc., Beijing, China, Technical Report, 2011.

[9] Rich Goyette. An analysis and description of the inner Workings of the FreeRTOS kernel." Course Report for SYSC5701: Operating System Methods for Real-Time Applications, Department of Systems and Computer Engineering, Carleton University. April 2007.

[10] Antonio Paoillo, Oliver Dersenfans, Vladimir Svoboda, Joel Goossens, Ben Rodriguez. A New configurable and parallel embedded real-time micro-kernel for multi-core platforms. In Proceedings of the ECRTS Workshop on Operating Systems Platforms for Embedded Real-Time applications (ECRTS-OSPERT'15), July 2015.

[11] Sukhyun Seo, Junsu Kim, Su Min Kim. An analysis of embedded operating systems: Windows CE, Linux, VxWorks, uC/OS-II, and OSEK/VDX. Journal of Applied Engineering Research, 12(18), pp. 7976-7981, 2017.

[12] C. M. Krishna and Kang G. Shin. Real-Time Systems (McGraw-Hill Series in Computer Science). McGraw-Hill College. December 1996.

[13] K.M. Mallachiev, Nikolay Pakulin, and Alexey Khoroshilov. Design and architecture of real-time operating system. Proceedings of the Institute for System Programming of the RAS. Vol. 28, pp. 181-192. 2016.

[14] Paul Leroux. RTOS versus GPOS: What is best for embedded development? Embedded Computing Design. January 2005.

[15] Niklas Holsti. Status of the Bound-T time and stack analyser. http://www.bound-t.com/.

[16] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: Min S.L., Pettit R., Puschner P., Ungerer T. (eds) Software Technologies for Embedded and Ubiquitous Systems. SEUS 2010. Lecture Notes in Computer Science, vol 6399. Springer, Berlin, Heidelberg, 2010.

[17] AbsInt. The Industry Standard for Static Timing Analysis. https://www.absint.com/ait/index.htm

[18] US-Cert, Vulnerability Note VU362332, Wind River Systems VxWorks debug service enabled by default, https://www.kb.cert.org/vuls/id/362332/

[19] Bonnie Zhu, Anthony Joseph and Shankar Sastry, A taxonomy of cyber attacks on SCADA systems. 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing, Dalian, pp. 380-388, 2011.

[20] Huan Yang, Liang Cheng, and Mooi Choo Chuah, Detecting payload attacks on programmable logic controllers (PLCs), IEEE Conference on Communications and Network Security (CNS), Beijing, China, May 2018.

[21] Mohammad Hamad, Zain A. H. Hammadeh, Selma Saidi, Vassilis Prevelakis, and Rolf Ernst. Prediction of abnormal temporal behavior in real-time systems. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18). ACM, New York, NY, USA, pp. 359-367. 2018.

[22] Bjoern Dusza, Christoph Ide, Liang Cheng and Christian Wietfeld, CoPoMo: a context-aware power consumption model for LTE user equipment, Transactions on Emerging Telecommunications Technologies, Vol. 24, No. 6, pp. 615-632, 2013.

[23] HV. Cheptsov and A. Khoroshilov, "Dynamic Analysis of ARINC 653 RTOS with LLVM," 2018 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russia, 2018, pp. 9-15, doi: 10.1109/ISPRAS.2018.00009.