A Study on the Portability of IoT Operating Systems

Renata Martins Gomes
renata.gomes@tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

ABSTRACT

The IoT is set to permeate our lives as a new and global super infrastructure, where billions of devices with an unprecedented variety of hardware architectures will interact. To enable IoT applications and services to run everywhere without major adaptation, operating systems (OS) provide standardized interfaces to the heterogeneous hardware. As a consequence, an operating system for IoT devices must be available for a huge number of target platforms, from low-end to high-end devices, and it must guarantee different levels of dependability (e.g., safety, security, real-time, maintainability) that each application will require. Some of these hardware architectures do already exist, others will emerge over time and introduce new or improved features that must be supported or exploited by the OS. In order to succeed, an OS must thus be portable, not only concerning its functionality, but also its verified dependability.

This paper tries to answer the question of how portable existing IoT OSs are, analyzing five popular OSs on their design, development, and testing processes, as well as the quality of available ports. We close with a suggestion on how to improve portability for future OS designs.

KEYWORDS

Portability, embedded operating systems, IoT

1 INTRODUCTION

An Operating System (OS) is expected to ease the development of applications by providing a unified execution environment on supported hardware, and relieving the application developer from the burden of performing micromanagement on limited system resources like memory, processor cores, execution time, etc. In this context, an OS is also responsible for ensuring the properly interleaved execution and interaction of concurrent tasks without bothering the developer with scheduling, synchronization, security or hardware-specific mechanisms demanding low-level knowledge about the hardware.

The Internet of Things (IoT) is set to permeate our lives as a new super-infrastructure, with applications ranging from monitoring our health to controlling our cars and managing our cities. Billions of devices are expected to interact in the IoT, and we can already see a growing variety of hardware architectures, communication stacks and programming paradigms, each specializing on specific purposes and different requirements or constraints. An OS that aims to become the Linux or Android for the IoT will need to be able to run on several platforms, ranging from low-end sensor nodes to high-end automotive electronic control units (ECUs), guaranteeing different levels of dependability [4] that each application will require. Thus, the OS must be portable, not only concerning its functionality, but also its verified dependability with respect to safety, security, real-time, and maintainability.

A *Port* is the realization of a software (here: OS) for a specific target environment. A Port can be created by *implementing* a specification (i.e., from scratch) or by *porting* (i.e., adapting) an existing Port for a new target environment. A piece of software is said to be *portable* if the effort of porting it is lower than the effort of implementing it from scratch [22]. In other words, the lower the porting effort, the more portable the software. For low-level (i.e., hardware-dependent) software, such as OSs and drivers, this environment is the hardware platform (e.g., the processor) they run on, and, due to different hardware configurations and features, just recompiling the source code is not enough to create a new port. Today, rewriting parts of the software is then unavoidable, and it is thus crucial for its longevity that the software architecture is designed such that the related effort is kept minimal. Other factors that affect the actual porting effort of low-level software are the level of expertise the developers have on both the software and the target platform, as well as the question to which extent the already supported platforms differ from the new target platform. Besides, for many systems in the IoT, the mere existence of executing code in a port is not enough: each port must guarantee to retain both functional behavior as well as non-functional properties for safety, security, real-time operation, maintenance, etc.

FGBS '21, March 11–12, 2021, Wiesbaden, Germany

https://doi.org/10.18420/fgbs2021f-01

This paper tries to answer the question of how portable existing IoT OSs really are, considering their design, development, and testing processes, as well as the quality of already available ports. It is organized as follows: Section 2 explains our evaluation methods and presents the portability aspects of each analyzed OS. In Section 3, we assess how portable we found the OSs to be, summarizing positive and negative aspects. We complement the OS analysis with a literature review on porting experiences and porting errors in Section 4. Finally, Section 5 draws a conclusion and gives a short look ahead on how OS portability could be improved in existing and future OS architectures.

2 PORTABILITY ASPECTS

In this section, we present the portability aspects of five open-source OSs commonly used in the IoT and related domains, such as Wireless Sensor Networks (WSN) and automotive. Since portability strongly depends on the software design and source code, our deliberate choice of open-source projects enables us to dig deep into each OS, instead of relying only on user-level information given by the providers. This study is based on our experience in both using and developing Real Time Operating Systems (RTOSs), and also considers, besides the source code, the website and available documentation of each OS.

In advance: a high number of supported platforms does not necessarily mean the OS is easily portable, nor does a low number indicate that it is difficult to port. First, supporting several similar devices requires much less effort than supporting very different devices, e.g., processors with completely different Instruction Set Architectures (ISAs). Also, the number of ports often depends on other factors, e.g., the time the OS had to mature, the size of its developer community, and the focus it gives to the amount of supported devices versus the quality of the ports (which is a common trade-off in software development).

This is not an exhaustive study of all IoT OSs, but a detailed study of some representative OSs:

- **FreeRTOS**[1] is one of the most used RTOSs in the IoT, and has ports to several platforms;
- **RIOT**[2] is another well known RTOS for the IoT, and also supports several platforms;
- **seL4**[3] is an example of a formally verified and secure RTOS;
- **Contiki-NG**[4] is an event-driven OS that is widely used in WSN, specially for low-end devices;

[1]FreeRTOS v10.2.1, released on 13th May 2019
[2]RIOT repository on 25th October 2019
[3]seL4 repository on 12th November 2019
[4]Contiki-NG repository on 14th November 2019

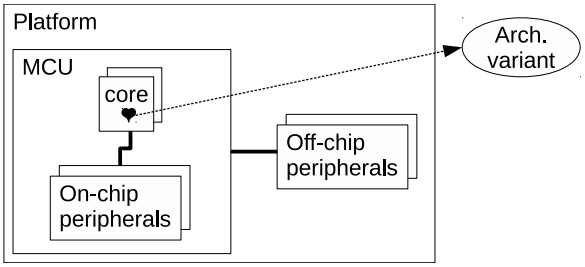

Figure 1: Relation between platform, MCU, cores and architecture variant.

- **ERIKA3**[5] is an OSEK/VDX implementation, mainly used in the automotive industry.

Table 1 summarizes the number of devices each OS currently claims to support. In order to unify the wording that is used by the different providers, we define that a *platform* is composed of *a Microcontroller Unit (MCU)* and attached off-chip peripherals. An MCU contains on-chip peripherals and one or more *cores*, each implementing an *architecture variant*, as shown in Figure 1. We consider an *architecture* as the specification of the instruction set, for example ARM, RISC-V, x86, etc, while an *architecture variant* is e.g., 32 or 64-bit versions of an architecture, or versions and variations, such as the ARM Cortex-M0, Cortex-M4, Cortex-A9, etc.

Table 1: Number of supported devices in each OS

OS	Arch.	Arch. variants	MCUs	Platforms
FreeRTOS	23	49	106	117
RIOT	7	15	39	167
seL4	3	10	20	23
Contiki-NG	5	6	8	28
ERIKA3	7	10	18	22

The portability aspects in this section try to assess how easy it is to port each of the OSs considering how the hardware abstraction layer is specified, how easily low-level code can be reused, and how ports are tested. We have tried to asses how their source code is organized and how it was ported so far, and whether common code, that avoids code replication, is available for similar platforms. We also surveyed the features of available ports, whether some are incomplete or unimplemented, and if there were functional and non-functional differences between them. Concerning testing or verification, we tried to asses how ported code is checked for completeness and correctness, and whether or how e.g., specifications or requirements are provided. The results of the portability aspects of each studied OS follow in the next subsections.

[5]ERIKA3 repository on 12th September 2019

2.1 FreeRTOS

FreeRTOS [1] is one of the most used open source RTOSs in the embedded domain [2], with the highest number of supported architectures and MCUs among the studied OSs. Developed since 2003, it is now owned by Amazon Web Services and distributed under the MIT license, meaning it can be used for any purpose without restriction.

The source code is organized in two main directories: `Source` contains the generic kernel code in its root, include files in a subdirectory, and low-level code in another, called `portable`. Interestingly, the `portable` directory has subdirectories for different compilers, instead of the classic separation by architecture or MCU. The implementation is in MCU directories inside the compiler directories. An exception is for the ARM Cortex variants, that are specific to each variant (e.g., Cortex-M0, Cortex-A9) but not to MCUs that implement them. MCUs with ARM7 or ARM9 have their own specific implementations, without any shared code between the ones with the same architecture variant. In fact, many ports are available for the same MCU or architecture variant on different compilers and still, no code is shared. It is instead mostly replicated, with differences limited to pragmas, casts that silence compiler warnings, and compiler-specific defines. In some cases, the code has a different structure, but its expected functionality is the same. There are "master copies" implemented for Cortex-M23 and Cortex-M33 (ARMv8-M) that are, prior to each release, copied into each corresponding compiler port. `Demo` contains demo applications with subdirectories for each platform-compiler combination. A lot of additional low-level code, such as board initialization, drivers, and even interrupt and watchdog handling are implemented inside the demos, at application layer, instead of a clear separation into reusable board support packages. While there is some code sharing for some tests, demos for some platforms on different compiler replicate large chunks of code. Similar platforms also have a considerable amount of replicated code.

The low-level kernel functionality is limited to context switching and some housekeeping, while other vital functionality of the kernel, such as initialization, clock, interrupt, and watchdog handling are outsourced to the demos. We also found evidence that the OS does not clean up memory allocated (by the kernel) to deleted tasks, nor guarantees that itself will run on supervisor mode, even if it will not work correctly otherwise.

There is no real specification for the low-level code, and according to the porting guide[6], one should simply copy two files to be ported from another existing port, delete function and macro bodies, and then implement the new port. Turns out that some ports have more than the two mentioned files, the functions to be implemented are not at all commented, and there are functions implemented for some ports that are not present on others. The porting guide also states that porting to a "completely different and as yet unsupported MCU is not a trivial task". In fact, there is even a company selling porting and testing services.

There are tests that "attempt to provide 'branch' test coverage" for generic kernel code, and some low-level code testing inside the demos, including a test for the context switches. The context switch tests seem to be manually written, and it is not clear if by the same or another developer. While such tests may reveal software bugs, it is not clear how efficient they really are.

2.2 RIOT

RIOT [3, 10] is an open source OS for the IoT that started in 2008 as an OS for WSN. It is supported by the Freie Universität Berlin, INRIA, and Hamburg University of Applied Sciences, and developed by a community composed of companies, academia, and hobbyists. Its microkernel architecture supports multi-threading with a fixed-priority scheduler that allows for soft real-time capabilities, Inter-Process Communication (IPC), and synchronization primitives. There is also support for some network stacks and external libraries.

The source code is organized with a clear distinction between hardware-dependent and hardware-independent code. The hardware-dependent code is separated in the directories `cpu`, `boards`, and `drivers`: `cpu` contains the implementation for the supported MCUs. It is organized such that common code for the same architecture or variant can be defined. The porting guide requires that common code is used in more specific ports, and that it is implemented as soon as a second platform of an existing architecture is introduced. `boards` contains configuration and initialization code for each supported platform, as well as a `common` directory where shared code can be found for similar platforms. `drivers` contains drivers for off-chip peripherals.

An interesting feature of RIOT is its driver concept: all off-chip peripheral drivers are platform agnostic, only communicating with the MCU via the so called Peripheral Driver Interface. Each MCU implementation must provide the driver interface for its on-chip peripherals. There are also Application Programming Interfaces (APIs) for common off-chip peripherals, such as for networking, CAN buses, ADC, etc., and a generic abstract layer for sensors and actuators.

In our analysis concerning the completeness of the available ports, we discovered several unimplemented features, and major and minor issues that lead to inconsistencies across different ports and may lead to critical bugs. We found cases where unimplemented functions return an error at runtime, but no documentation informing about it could be

[6] https://www.freertos.org/FreeRTOS-porting-guide.html

found. For example, the common code for Cortex-M CPUs implements the Memory Protection Unit (MPU) configuration function only for the Cortex-M23 variant, while for other variants, it will silently fail. Another example is the real-time clock driver for the esp8266 MCU, which is completely unimplemented, and it is not clear from the documentation how to avoid running into errors when using this platform. Other issues include comments in the source code, such as "this implementation needs major rework", "this file is incomplete, (…) there are some inconsistencies throughout the family which need to be addressed", "generalize to handle more timers and make them configurable", "only disable watchdog on debug builds".

RIOT uses the Murdock Code Integration framework that executes static, unit, and compile tests automatically for several build configurations, besides functional tests on selected platforms. The code base also has a `tests` directory with several tests, some of which can be performed automatically. Well defined processes for the developing community also try to ensure high code quality and solid documentation in the code base.

2.3 seL4

The seL4 microkernel [12] is an open source RTOS designed for security and high performance, and developed with the aim for formal verification. It is developed, maintained, and formally verified at Data61 by the Trustworthy Systems Group (formally NICTA) and owned by General Dynamics C4 Systems. The seL4 is the most advanced member of the L4 family, which is developed since 1993, and is a general-purpose microkernel for application areas that target security and reliability and run on embedded systems with a Memory Management Unit (MMU).

The seL4 is composed of several repositories containing parts of the project, such as the kernel, its specification and proofs, test and benchmarking suites, etc. We have mainly analyzed the kernel repository, while considering the others when assessing seL4's verification, testing, and benchmarking.

The kernel's repository [21] has C and assembly source code in the directories `include` and `src`, as well as C bindings for the kernel's Abstract Binary Interface (ABI) in the directory `libsel4`. Their subdirectories are named or prefixed with `arch` if they contain architecture-specific code and `plat` if they contain platform or MCU-specific code. If the code is hardware-independent, the subdirectory name only reflects the OS module it contains. Those subdirectories with hardware-specific code have further subdirectories to reflect the target hardware. Most have one subdirectory for each supported architecture (ARM, RISC-V, and x86), and further variants of those architectures (such as 32/64 bits or ARM versions) are implemented within subdirectories. Common code is always as close to the root as possible, meaning it can easily be reused by more specific ports. While the kernel only has drivers for serial communication and a timer, other device drivers and libraries are provided outside the trusted computing base and run in user space. They are in the device-specific files in `drivers`, and are related to the respective platform with a device tree scheme.

Verification in seL4 uses formal mathematical proof in the theorem prover Isabelle/HOL. The fully verified ARM port on the Sabre Lite board is proved to be functionally correct from the C implementation down to the binary, i.e., there is proof that the C implementation adheres to its specification, and that the binary code is a correct translation of the C code. It also has proofs that security properties (integrity and confidentiality) are enforced by the specification (as long as the specification is used properly), and guarantees hard real-time with a sound and complete timeliness analysis. On other platforms, with ARM and x86-64 architectures, the functional correctness proof is complete, while it is ongoing for the RISC-V port. There is no information on when other proofs will be available.

2.4 Contiki-NG

Contiki started in 2002 as a light-weight OS for WSN. In 2017, a fork of Contiki, now known as Contiki-NG [8], was started focusing on reliable and secure communication, modern IoT platforms, and more modern and agile structure and development processes. Contiki-NG aims at resource-constrained devices in the IoT, is open source and community driven, and comes with a BSD license that allows the free use and distribution of the code, as long as the license is retained in the source code.

Contiki-NG only supports cooperative multitasking, which reduces its complexity and memory footprint, but transfers some of the multi-threading burden to the application layer. While a process never preempts other processes, interrupts might do so, and care must be taken with code that may run in interrupt context, since some system and library functions are not interrupt context safe. Also, according to the wiki[7], the "(application) developer must make sure that processes do not keep control for too much time and that long operations are split into multiple process schedulings".

The source code has a generic OS code directory, while `arch` contains hardware-dependent code. Inside it, `dev` contains drivers for e.g., sensors, transceivers, and disks. Another directory, `platform`, has platform-specific code and configuration. Similar platforms have replicated code, and there is no shared code between them. Code specific to architectures

[7] https://github.com/contiki-ng/contiki-ng/wiki

and MCUs is on the same level inside cpu. The MSP430 architecture implementation shares code with the supported MCUs, which are implemented inside the architecture's directory. ARM, on the other hand, has an ARM architecture directory with support for Cortex-M3 and Cortex-M4 cores, with shared code, and another three directories supporting Cortex-M3 or Cortex-M4 MCUs, without any shared code.

Contiki-NG's porting guide is rather complete, clearly stating what must be supported for MCUs and for platforms, and there is a well defined API for what must be implemented when creating a new port. On the existing ports, we only found few unimplemented features, all documented, and some features still to be implemented on drivers.

The source base provides several example and test applications. One benchmark example on communication runs on a testbed and results are posted online[8]. To ensure non-regression, the available tests are automatically run on a continuous integration service, called Travis, for every pull request and merge.

2.5 ERIKA3

Erika Enterprise [9] is an open source RTOS from Evidence Srl for the automotive domain. Development started in 2000 aiming to be an open source OSEK/VDX OS that could be used both by industry and academia, in research, development, and production. ERIKA v2 is OSEK/VDX-certified on an ARM Cortex-M4 and on the Infineon Tricore 26x MCUs. From 2014, the OS started to be completely reimplemented, giving birth to ERIKA3. The new OS still implements OSEK/VDX, but is not yet certified. It focuses on proper multicore support and is designed to eventually implement the AUTOSAR OS specification. Now it also offers a better licensing scheme: the free GPLv2 license allows users to link their own code, as long as it is also made open source, while paid licenses can be purchased by companies willing to have ERIKA3 in their products. Erika Enterprise is open source, however its development is not community driven: Evidence Srl is the main developer, and some companies contribute code for their own platforms. External contributors must sign a contribution agreement.

The source code is located in the pkg directory in the source base. Inside it, common contains shared code and includes for tooling and compilers. The OS generic code is in kernel, while generic code for context switching and hardware abstractions is in std. Hardware-dependent code is located in arch. Inside it, each supported architecture variant has a corresponding directory with code for e.g., context switching and interrupt handling, as well as directories for the supported platforms and example applications. The separation between MCU and architecture variant code is not consistent: some variants have MCU directories for the more specific code, while others have MCU-specific files in their root. Another example of this inconsistency is that code related to timers is in architecture variant files with MCU-specific defines, instead of explicitly configured within MCU code. Platform-specific code can only be found in some of the given examples, where code is often replicated with few differences, if any.

In order to provide a stable working environment for users, regression tests are automatically executed in a Jenkins environment, guaranteeing that the code keeps working on a set of boards.

In our analysis concerning the completeness of the available ports, we discovered some unimplemented functions, and potential critical bugs, such as untested sign extensions. Some features that should be configurable are hard-coded on some ports, and some of the ports do not yet support multicore. Commercial technical support offers ports to new MCUs.

3 ASSESSMENT

In this section, we discuss our findings from Section 2, assessing how portable the analyzed OSs are, and highlighting the main strengths and weaknesses to their portability.

Despite supporting 23 different architectures, we see several hindrances to **FreeRTOS**'s portability. First, the separation of code by compilers, instead of by architectures, seems odd, and it is not clear why it would be so central in the code base. On the one hand, it might ease the user's first contact with the OS and the toolchain they shall use. On the other hand, it makes the code base much harder to maintain, and changes on the low-level software are hard to deploy on all affected ports. Additionally, the outsourcing of vital OS functionality into the application layer (see "demos" in the code) hides a big part of the actual effort of porting the OS. The lack of specification on the low-level kernel and on other relevant functionality also requires a developer to self-investigate existing code in order to understand and port the OS. The high number of architectures that FreeRTOS supports might be explained not due to the OS's portability itself, but by the focus given on supporting as many platforms as possible, together with over 17 years of development time and the fact that hardware providers themselves work together with FreeRTOS developers to create ports for their devices.

RIOT is very well structured to support easy portability through code separation, specification of what must be ported, as well as specific APIs for driver communication, making the code reusable and relatively easy to port. Its solid test system also positively contributes to its portability and quality. Allied with the large development community, they manage to have the highest number of supported platforms,

[8] https://contiki-ng.github.io/testbed

and to support seven completely different architectures. Despite advantages on portability and their quality-striving approach, there are still many open issues (see "todos" in the code) to be addressed in many ports, which show that porting is still not a straightforward task.

While **seL4** only supports three architectures, we find its approach to formal specification, along with its well-structured code, very positive for its portability. Its verification, testing, and benchmarking assures quality and correctness of the ports, and makes it the only OS to seriously go beyond functional properties. The related complexity affects the trade-off on how many ports are already available and explains the limited number of supported platforms.

Contiki-NG is a simple OS and supports simpler platforms when compared to the other OSs we have analyzed. This likely reduces the effort of porting it. Nevertheless, there are few supported MCUs, and inconsistencies in the code separation leave room for improvement. Still, the ports that are available seem to have high quality and the sound processes of testing and benchmarking along with the large development community are positive to Contiki-NG's portability.

ERIKA3 is a rather complex OS, created to support high-end devices and implement complex functions for the automotive domain. This increases the effort of porting it. We believe, however, that the code base can be made more portable through a clear separation of hardware-specific code and configurations. The testing process could be extended, and a bigger development community would probably improve its portability as well.

4 PORTING EXPERIENCES IN THE LITERATURE

In addition to analyzing some existing OSs, we have also searched the literature for experiences on porting and for studies that analyze the (bad) consequences of porting errors.

Among the OSs we have analyzed, Oikonomou and Phillips [23] resume the port of Contiki to Sensinode devices, which had been discontinued, and reveals problems with choosing the ideal memory model and stack sizes. They show that a port needs to consider not only the ISA, but also compilation and memory model relations, code size versus code simplicity, stack sizes, etc. Also porting Contiki, Stan [26] reports difficulties in porting the low-level code, even though it represents only a little part of what had to be ported. Zhang *et al.* [29] have ported an AUTOSAR-compliant OS, similar to ERIKA3, to a Raspberry Pi. The port is still missing some functionality, and they needed inspiration from other OSs in order to understand how to write an OS for their target.

Of course, this is not an exclusive problem of these OSs. Even Linux and UNIX, generally considered to be portable, pose challenges: back in 1984, Bodenstab *et al.* [5] found UNIX to be "extremely portable", but pointed out that several software changes were needed to account for hardware differences between the base and the new targets, even though the base port was selected based on the similarity its target had with the new target. An entire section is dedicated to the effects of hardware architectures on porting. More recent works show that the problem remains, impacting porting of Linux from x86-32 to x86-64 [18], and to the Renesas M32R processor [27], where kernel headers with extensive use of assembly statements were "very difficult to port" and their "insufficient and inadequate rewriting easily caused very hard to debug problems".

Even on the application side, especially for embedded systems, porting often becomes difficult, as Horman [14] shows. Lewis [19] proposes MetaH, a tool to improve porting of avionic application software that provides target-independent timing semantics. MetaH tackles problems caused by timing dependencies, complex processor architectures, and specialized devices that drive software integration and maintenance costs up. Smith *et al.* [25] propose a framework to make robotic software more portable, since its high specialization results in an "almost non-existent level of software portability".

Several works have already shown that a significant number of bugs come from porting errors and described their consequences for the overall quality of OSs. Chou *et al.* [7] found a high rate of errors in the drivers, listing as possible explanations that the driver developer is usually less familiar with the OS than the kernel developers, and that drivers are not as heavily tested as the kernel. Li *et al.* [20] identify errors in Linux and FreeBSD resulting from developers forgetting to rename identifiers after porting code. In [24], Ray *et al.* confirm these findings with an empirical study of porting errors in Linux and FreeBSD, where they also found other common error categories and presented a "semantics porting analysis algorithm". Code duplication, common practice when porting code, is another common source of bugs. In a case study of bugs related to duplicated (cloned) code, Juergens *et al.* [17] discovered that "nearly every second, unintentional inconsistent changes to clones lead to a fault". Gabel *et al.* [11] discuss the difficulty to maintain duplicated code that is supposed to evolve together when bugs are fixed or new features are introduced. In [16], Jiang *et al.* discovered many previously unknown bugs in the Linux kernel and Eclipse, confirming that code duplication is error-prone.

The source of these problems when porting code might be found in the porting process. Cho and Bae [6] have found that typical software development processes do not work for the porting process, since often the documentation and methods available are not helpful for the developer. They propose a new process that tries to tackle this, improving

communication between developers and enhancing the documentation used when porting. Hu *et al.* [15] also reckon the problem with the porting process. They tried to solve it by synthesizing an OS instead, and report on the problems found when trying to do so, such as scalability of models, solvers and synthesis, as well as the difficulty of modeling low-level functionality in a machine-independent way.

From our own and other's experiences we also suggested to replace manual coding (according to a specification) by formal modeling [13]. This paradigm shift in the development process allows to first model the OS functionality on an abstract hardware, and then instantiate it to different targets. Significant parts of the model can thus be reused, even when modeling low-level OS functionality. The final instantiations can then be used to both verify for correctness and automatically generate the executable OS code.

5 CONCLUSION AND OUTLOOK

During this study, we have seen that porting an embedded OS is not a trivial task, and that it is a common source of bugs. Beyond the code structure, several other factors impact on how portable and reusable existing OS code can be. Well-structured testing and benchmarking improves the code quality of each port, and open source releases can gather a large developer community to further increase the quality and quantity of ports. However, we have seen that this is not enough. Especially in contexts where dependability must be guaranteed, such as the IoT, software development must go beyond intense testing and a large work force of programmers. Formal methods greatly ensure software quality and dependability [28], and seL4 shows that it is feasible to model and verify significant parts of an OS.

We also continue to integrate and establish formal methods in the software development process to support verification and reduce manual implementation effort. Our code generator that is currently under development, automatically generates target-specific code from instantiated OS models, keeping the ports for different target architectures consistent when the model changes.

A challenge that remains is scalability with respect to the code and model size, as formalizations and proofs will demand much effort for feature-rich OSs. We still believe that it is worth the effort and that processes and training on formal methods must become standard in embedded software development. In fact, formal modeling can be combined with code generation to further reduce manual implementation effort and simplfiy portability, guarantee dependability, and improve the overall quality of embedded OS in general.

REFERENCES

[1] Amazon Web Services. [n.d.]. FreeRTOS. https://www.freertos.org/ Accessed: 2021-01-23.

[2] AspenCore. 2019. Embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[3] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. 2018. RIOT: an open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (Dec 2018), 4428–4440. https://doi.org/10.1109/JIOT.2018.2815038

[4] Carlo Alberto Boano, Kay Römer, Roderick Bloem, Klaus Witrisal, Marcel Baunach, and Martin Horn. 2016. Dependability for the Internet of Things—from dependable networking in harsh environments to a holistic view on dependability. *e & i Elektrotechnik und Informationstechnik* 133, 7 (2016), 304–309. https://doi.org/10.1007/s00502-016-0436-4

[5] D. E. Bodenstab, T. F. Houghton, K. A. Kelleman, G. Ronkin, and E. P. Schan. 1984. The UNIX system: UNIX operating system porting experiences. *AT T Bell Laboratories Technical Journal* 63, 8 (1984), 1769–1790.

[6] D. Cho and D. Bae. 2011. Case study on installing a porting process for embedded operating system in a small team. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion*. 19–25.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct 2001), 73–88. https://doi.org/10.1145/502059.502042

[8] Contiki. [n.d.]. Contiki-NG. https://github.com/contiki-ng/contiki-ng Accessed: 2021-01-28.

[9] Erika Enterprise. [n.d.]. ERIKA3. https://www.erika-enterprise.com/ Accessed: 2021-01-28.

[10] FU Berlin. [n.d.]. RIOT - the friendly operating system for the internet of things. https://www.riot-os.org/. https://www.riot-os.org/ Accessed: 2021-01-23.

[11] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. 2010. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA10)*. Association for Computing Machinery, New York, NY, USA, 175–190. https://doi.org/10.1145/1869459.1869475

[12] General Dynamics C4 Systems. [n.d.]. The seL4 microkernel. https://sel4.systems/ Accessed: 2021-01-25.

[13] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2020. A formal modeling approach for portable low-level OS functionality. In *Software Engineering and Formal Methods*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 155–174.

[14] Neil Horman. 2009. Porting to Linux the right way. In *Proceedings of the Linux Symposium*, Robyn Bergeron (Ed.). The Linux Foundation, 45–56. https://www.kernel.org/doc/mirror/ols2010.pdf

[15] Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. Trials and tribulations in synthesizing operating systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS19)*. Association for Computing Machinery, New York, NY, USA, 67–73. https://doi.org/10.1145/3365137.3365401

[16] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE âĂŹ07)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/1287624.1287634

[17] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. 485–495.

[18] Andi Kleen. 2001. Porting linux to x86-64. In *Proceedings of the Linux Symposium*.

[19] Bruce Lewis. 2002. Software portability gains realized with METAH and Ada95. In *Proceedings of the 11th International Workshop on Real-time Ada Workshop* (Mont-Tremblant, Quebec, Canada) *(IRTAW '02)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/584418.584423

[20] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: a tool for finding copy-paste and related bugs in operating system code.. In *OSdi*, Vol. 4. 289–302.

[21] Anna Lyons, Adrian Danis, Yyshen, Hesham Almatary, Stephen Sherratt, Amirreza Zarrabi, Kent McLeod, Gerwin Klein, Latent Prion, Joel Beeren, Thomas Sewell, , Adam, Rafal Kolanski, Alexander Boettcher, Partha Susarla, Matthew Brecknell, Jeff Waugh, Seb Holzapfel, Christopher Guikema, Corey Richardson, Cloudier, Robbie VanVossen, Mktnk3, Mokshasoft, Tim Newsham, Luke Mondy, Jesse Millwood, Jsuann, Nathan Studer, and Curtis Millar. 2018. seL4/seL4: MCS pre-release. https://doi.org/10.5281/ZENODO.591727

[22] J. D. Mooney. 1990. Strategies for supporting application portability. *Computer* 23, 11 (1990), 59–70.

[23] G. Oikonomou and I. Phillips. 2011. Experiences from porting the Contiki operating system to a popular hardware platform. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*. 1–6.

[24] B. Ray, M. Kim, S. Person, and N. Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 367–377. https://doi.org/10.1109/ASE.2013.6693095

[25] R. Smith, G. Smith, and A. Wardani. 2004. Software reuse in robotics: enabling portability in the face of diversity. In *IEEE Conference on Robotics, Automation and Mechatronics, 2004.*, Vol. 2. 933–938. https://doi.org/10.1109/RAMECH.2004.1438043

[26] Alexandru Stan. 2007. Porting the core of the Contiki operating system to the TelosB and MicaZ platforms. *International University, Bremen* Bachelor thesis (2007).

[27] Hirokazu Takata, Naoto Sugai, and Hitoshi Yamamoto. 2003. Porting Linux to the M32R processor. In *Linux Symposium*, John W. Lockhart (Ed.). The Linux Foundation, 398.

[28] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal Methods: Practice and Experience. *ACM Comput. Surv.* 41, 4, Article 19 (Oct. 2009), 36 pages. https://doi.org/10.1145/1592434.1592436

[29] Shuzhou Zhang, Avenir Kobetski, Eilert Johansson, Jakob Axelsson, and Huifeng Wang. 2014. Porting an AUTOSAR-compliant operating system to a high performance embedded platform. *SIGBED Rev.* 11, 1 (Feb. 2014), 62–67. https://doi.org/10.1145/2597457.2597466