The Case for an Opinionated, Theory-Oriented Real-Time Operating System

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

ABSTRACT

While today it is *in principle* possible to construct cyber-physical systems in a temporally sound way, *in practice* this rarely happens because, with the current real-time operating system (RTOS) foundations, the prerequisite investments in time, expertise, and resources are prohibitive. This position paper argues that this is due to a lack of *(i)* easy-to-use, high-level abstractions in contemporary RTOSs and *(ii)* first-class support for adaptive systems provisioned at below-worst-case levels. A fundamentally different RTOS concept that seeks to address these issues, based on the novel notion of *theory-oriented RTOS design*, is introduced and a concrete manifestation, called TOROS, currently under development is sketched.

1 INTRODUCTION

In terms of scientific progress, the real-time systems community can look back on an extraordinarily successful couple of decades; the community's collective understanding of time-critical computing has come an impressively long way since the field's early beginnings in the 1970s and the community's concrete formation in the 1980s, with major advances in both the foundational theory and analysis as well as the practical engineering of real-time systems. As a result, developers of cyber-physical systems (CPSs) today can choose among a diverse selection of battle-tested real-time operating systems (RTOSs) and leverage a rich and sophisticated formal foundation that enables, at least in principle, even large and complex systems to exhibit predictability at historically unprecedented scales. It is no surprise, then, that state-of-the-art real-time methods and results have been adopted by leading corporations in many demanding application domains, including avionics, automotive systems, and factory automation (among others). Inarguably, this is in many ways a great success story.

The picture, however, looks considerably less rosy if one steps outside of domains dominated by large corporations that can afford substantial research and development (R&D) departments, which usually retain specialized staff and in-house technology consultants, including real-time systems experts. Especially when narrowing the focus to the adoption of *temporally sound* system architectures and *analytically sound* timing analysis methods, the successful transfer of real-time systems research into practice appears to be much less common overall, and certainly much less common than is desirable.

For instance, among the many small- to medium-sized companies (or development teams) that download, configure, and deploy Linux with the `PREEMPT_RT` patch, what is the fraction of companies/teams that also employ some variant of formal schedulability analysis? Similarly, among the thousands of research and product teams that develop robotics applications using the popular ROS middleware (which commonly runs on Linux), what is the fraction of teams that deploy ROS using proper real-time policies that guarantee predictable, analyzable resource allocation (such as reservation-based scheduling with `SCHED_DEADLINE` in Linux)? At least anecdotally, the answer to both questions is unfortunately "close to zero." In these areas, as in many others, temporally sound methods are simply not mainstream (yet), and there presently appears to be little momentum towards change in the status quo.

In other words, beyond the well-known, large, and technologically advanced users of state-of-the-art real-time technology (such as Airbus, Thales, ABB, Bosch, and the like), there exists a long and heavy tail of much smaller companies and research groups that *could* and *should* benefit from proper, analytically sound real-time system design, predictable software architectures, and formal methods for verification and validation, but that currently do not.

This state of affairs is regrettable, and I posit that reaching the large number of potential users in the long tail—users who without a doubt care deeply about the timeliness of their systems, but who are neither formal timing experts themselves nor supported by R&D departments with on-staff real-time expertise—represents the biggest challenge to, and opportunity for, continued growth, relevance, and real-world impact of the real-time systems community. Case in point, it is difficult to see how the widespread proliferation of autonomous vehicles and service drones imagined by many to become a hallmark feature of the 21$^{\text{st}}$ century—all of them safety-critical and operating in public space—is going to happen without order-of-magnitude improvements in certification costs. As of today, the best hope for such improvements is the pervasive use of formal methods, and especially so in the context of temporal correctness.

The core problem is not a fundamental lack of capabilities, but rather one of accessibility. The existing large body of literature on real-time systems holds the answers to many (or even most) of these potential users' timing problems, but this information is spread across thousands of papers, with at times unclear recommendations and often incompatible policies and assumptions, stemming from various schools of thought (*e.g.*, fixed-priority vs. earliest-deadline first scheduling, global vs. partitioned multiprocessor scheduling, to name a few). Navigating this literature is by no means an easy task to the uninitiated, and requires a substantial amount of time even in the best of circumstances—time that engineers in practice rarely have (if ever). Thus, my question: how do we make this treasure trove of scientific progress and insights *useful* and *accessible* to the development community at large, without requiring each and every user to earn an advanced degree in real-time systems first?

2 HURDLES TO ADOPTION

To achieve widespread adoption, I conjecture that it is necessary to fundamentally rethink how RTOSs are designed, with the goal of completely replacing the interfaces and abstractions that they offer to developers with simpler, higher-level alternatives. To motivate the design sketch presented in Sec. 3, I briefly review in the following what I consider to be major challenges to adoption today.

NGOSCPS '19, April 15, 2019, Montreal, Canada
.

Design-for-analysis is uncommon. Contemporary formal analysis requires the system to be designed *for* such analysis. Given the challenges inherent in obtaining provably sound formal analysis, it is simply unreasonable to expect it to work for just any given system with a structure not intentionally chosen to facilitate formal reasoning, which is true both for functional verification (*e.g.*, see seL4 [2]) and the verification of non-functional properties such as temporal correctness. This, however, represents a monumental barrier to adoption because it means that system architects and developers must be convinced to rely on formal analysis even *before* the system is designed. Conversely, even if a developer or system integrator later realizes that analysis support would have been beneficial, it is too late—at that point, the high cost of rearchitecting the system to enable formal analysis likely outweighs the benefits, or there is simply no time left in the project schedule to do so.

Furthermore, even if the system designers and developers are in principle interested in making their system analysis-friendly, there is a good chance that the actual design and implementation will fail to achieve this goal, due to the next issue.

Expertise barrier. Current RTOSs lack high-level abstractions, which renders temporally sound system design too costly in terms of effort, time, and expertise to be a common option in practice.

On the one hand, RTOSs commonly used in CPSs today—in particular, various real-time variants of Linux and specialized OSs like FreeRTOS, RTEMS, VxWorks, QNX Neutrino, or Nucleus RTOS—have accumulated a bewildering array of APIs, few of which are temporally sound. The root of the problem is that these interfaces expose *low-level mechanisms that are too difficult to use and combine correctly*, rather than predictable high-level abstractions. While low-level APIs like scheduling priorities, processor affinity masks, various kinds of semaphores, signals, pipes, sockets, and so on *could in theory* be used to realize predictable, analytically sound systems *if combined in just the right way*, their correct use requires a deep understanding of the relevant real-time theory and a careful selection of which interfaces to use and which to avoid. However, domain experts typically have little background in real-time scheduling theory—*nor should they be expected to have any!*

On the other hand, modern research RTOSs such as Composite OS [4], Fiasco.OC [3], or seL4 [2] emphasize policy freedom as a key design goal, such that the resulting OS can be used to implement *any* policy—which again requires the users of the OS to make all key policy choices, which (in the long tail) they are ill-equipped to make.

Separate tools and extra steps. Next, even if developers are not deterred by the above issues, actually integrating formal analysis tools into the development, integration, and testing and validation workflow is not at all easy. Besides the fact that most developers loath introducing "yet another" build dependency, available tool support is actually rather limited and not necessarily all that user-friendly. In fact, it is not uncommon for major vendors of static timing analysis tools to dispatch application engineers to customers to help integrate their (arguably somewhat arcane) tooling into the customer's build system and workflows on a case-by-case basis.

Needless to say, this situation does not further widespread adoption, especially with small development and research teams who rely primarily on open-source technology stacks and tool chains (*e.g.*, Linux, ROS, GNU binutils, *etc.*). In contrast, just imagine that Linux would provide for each real-time process a virtual file `/proc/$PID/max-response-time-estimate` that gives a built-in, up-to-date, *analytically sound* response-time bound based on current process priorities and the maximum arrival rates and execution times observed so far (*i.e.*, the analysis would be provably sound, but the inputs traced and thus readily available but uncertain). It stands to reason that such an easy-to-use facility would be quickly adopted and widely used during development, regression testing, component validation, integration testing and for runtime monitoring of Linux-based real-time systems—it would certainly go a long way towards popularizing response-time analysis. Nonetheless, it might still not be all that useful, due to the following mismatch.

Adaptive, below-worst-case provisioning. The main body of classic real-time scheduling theory makes two limiting standard assumptions: that the workload is *static* (*i.e.*, tasks do not leave or join the system, or change requirements), and that safe bounds on all *worst-case execution times* (WCETs) are known. Both are problematic in a modern CPS context. First, many CPS workloads are *inherently dynamic*. For instance, in an autonomous CPS such as a service drone, the runtimes of vision, planning, and mapping algorithms all depend on the environment. Further, different missions and situations naturally require different features or subsystems to be active. On-demand *adaptation* is thus essential. In particular, the system must reconfigure when faced with unforeseen load changes since it is not feasible to anticipate the worst-case resource needs for complex, environment-dependent CPS workloads. The essence of real-world engineering is graceful degradation, not static worst-case guarantees that are established once and then hold "forever."

Second, on modern commodity multicore processors, even just the difference between the 99^{th} percentile execution cost and the observed maximum can easily span an order of magnitude due to inherent hardware unpredictability. It is simply not possible in practice to provision all code based on observed maxima (*e.g.*, $10\times$ above the *typical* case), let alone based on safe, but pessimistic WCET bounds (if even available). Thus, in reality, most systems are not provisioned at worst-case levels. Instead, engineers exploit the fact that not all tasks exhibit maximum resource demands at the same time. However, this common practice is a fundamental mismatch with existing analysis, and a source of uncertainty that in practice is typically dealt with by blind over-provisioning—if there is "enough" extra capacity on average, it "probably" works out most of the time. But how much spare capacity is really needed? Today, this is largely left to guesswork, testing, and trial-and-error.

In summary, with the current analysis and RTOS foundations, it is still too difficult and too costly to construct CPSs such that their temporal correctness can be rigorously and efficiently ascertained—that is, without requiring a "valiant effort" on behalf of the developers that is uneconomical for all but the most critical CPS.

3 TOROS: A THEORY-ORIENTED RTOS

To attack the challenges laid out in the previous section, I have started a new clean-slate, from-scratch RTOS project called TOROS. TOROS is an experimental, exploratory, *theory-oriented* (and quite opinionated[1]) RTOS design targeting a long-term horizon, and seeks

[1] https://stackoverflow.com/questions/802050/what-is-opinionated-software

to fundamentally rethink the role of an RTOS without regard for legacy code, standards compliance, or existing RTOS conventions. The design is guided by the following principles and goals.

Theory-oriented RTOS design. To eliminate intrinsic and accidental unpredictability, *all provided OS abstractions must be temporally sound* and supported by sound analysis. *All others must be removed*, which includes leaving behind the classic (but analytically challenging) concept of long-running "processes."

To reiterate, the central design goal is to enable the construction of predictable, analytically sound multiprocessor real-time systems by developers who are not aware of any underlying theory, or its limitations and pitfalls. This means that *all primitives and abstractions offered by the RTOS must align with backing scheduling theory* such that any possible use, any possible combination of primitives can be automatically analyzed. In other words, accidental unpredictability must be virtually impossible (*i.e.*, well-behaved, idiomatic TOROS applications must not accidentally become unanalyzable).

Declarative, high-level RTOS abstractions. To make predictability affordable (*i.e.*, to make the design of predictable applications more approachable to non-real-time experts), the OS must provide (exclusively) *high-level, declarative interfaces* that allow users to state automatically checkable timing and resource-allocation goals.

Further, the OS should be opinionated software: rather than exposing many options and forcing application developers into making difficult policy decisions, the OS should implement policies that work well in most cases and offer "one right way" to realizing common real-time functionality. In the interest of simplicity, and in contrast to microkernel philosophy, TOROS aims for "freedom from choice" rather than keeping the kernel free of policy.

Temporal reflection. Since the system is expected to be provisioned below worst-case needs, and to expose real-time analysis to non-expert users, it must *transparently and continuously self-assess its temporal correctness* to warn and *proactively adapt* when assumptions are violated or declared timing goals can no longer be guaranteed. By incorporating sound, always-on schedulability analysis into the runtime system (instead of seeing it purely as an offline tool), and by continuously monitoring actual resource demands, the system will be able to *predict worst-case response times before they occur.* That is, as soon as changes in the load are observed that imply that the envelope of safe system states has been left (*i.e.*, that timing errors have become *possible*), the system can proactively adapt *before* a timing error actually occurs (in application-specific ways, *e.g.*, by disabling optional functionality, rerouting resources from less critical to more critical subsystems, *etc.*).

Structured uncertainty management. To allow the system to be provisioned below worst-case levels without completely giving up on analyzability, TOROS will rely heavily on slack reclamation [1], declarative slack pools, and a new kind of *correlation-aware sensitivity analysis* that takes typical execution costs into account to provide strong *probabilistic timing guarantees* at economically viable provisioning levels. Importantly, the result of this sensitivity analysis will be an estimated safety margin relative to the declared end-to-end timing goals (*i.e.*, a result of the form "an increase in execution time by $X\%$ has no ill effects with probability at least Y"), rather than a conventional yes/no schedulability judgement.

In particular, it will be possible to bound the *expected safety margin* (*i.e.*, the amount of slack available in the expected case). This approach reconciles existing practice (provisioning below the worst case, a necessity in practice) with a sound theory of why it works (with very high probability). Conversely, it will allow the system to trigger proactive adaptation in case the residual risk exceeds a given threshold (*i.e.*, if more slack is needed to stay below the risk threshold). That is, rather than relying on blind over-provisioning, TOROS will allow developers to tailor resource use in an informed way to reach any given level of acceptable risk.

While tracing is often frowned upon as it cannot establish a sound WCET, TOROS is fundamentally different: it will *trace correlations and percentiles*, and *not* WCETs, which is feasible since correlations and percentiles can be established with high confidence given a relatively modest number of samples.

Implementation sketch. To guarantee analyzability, the TOROS design retires the classical notion of long-running processes in favor of three (and only three) orthogonal concepts: *ephemeral jobs* (EJs) with run-to-completion semantics (*i.e.*, sequential execution contexts *that cannot suspend*), passive *logic and data compartments* (LDCs), and *guaranteed processor shares* (GPSs). LDCs are protection domains (*i.e.*, address spaces) and closely resemble the components in Parmer's Composite OS [4] and are in fact directly inspired by them. GPSs are the basic time abstraction and offer a guaranteed *long-term utilization* with a guaranteed *maximum latency.* Each EJ is associated with exactly one LDC and one GPS. Key to the envisioned theory-oriented design is that TOROS allows only two operations to compose jobs: *asynchronous invoke* (create a new EJ), optionally with barrier semantics (last to invoke creates a new EJ), and *synchronous invoke* (create a new EJ and wait for it to finish with call-return semantics). In particular, there is no "block," "wait," or "sleep" API—it will not be possible to reference an already existing job and wait for it in any way, or to suspend it, which naturally forces an *event-driven* (or *reactive*) programming model. Another innovation is to introduce *declarative occupancy constraints* to hide the challenging multiprocessor *real-time mutual exclusion* problem from application developers by treating each LDC as a monitor.

4 OUTLOOK

The TOROS project officially started in January 2019 and is kindly supported for the next five years by a Starting Grant of the European Research Council (ERC). As such, the project is still in a very early phase, open to design adjustments, and welcomes collaboration; I am particularly interested in constructive criticism of the sketched design and alternatives for achieving the stated goals.

REFERENCES

[1] M. Caccamo, G. Buttazzo, and L. Sha. 2000. Capacity Sharing for Overrun Control. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS).* 295–304.

[2] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP).* 207–220.

[3] A. Lackorzynski and A. Warg. 2009. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems.*

[4] Gabriel A. Parmer. 2010. *Composite: A component-based operating system for predictable and dependable computing.* Ph.D. Dissertation. Boston University.