Revisiting the Design of Systems for High-Confidence Embedded and Cyber-Physical Computing Environments

Richard West and Gabriel Parmer

{richwest,gabep1}@cs.bu.edu Tel: 1-617-353-2065

1 Introduction

As the complexity of emerging real-time and embedded software systems increases, new challenges beyond those focusing solely on timeliness guarantees are becoming increasingly significant. It is expected that embedded devices such as mobile phones and personal digital assistants will support tens of millions of lines of code in the foreseeable future. Web services, video-on-demand and multimedia databases are already appearing on hand-held devices and so it makes sense that software complexity will only increase over time. Avionics, automotive and medical application domains clearly dictate the need for software dependability in addition to traditional goals of timeliness constraints. Should unforeseen faults during software execution occur, it is paramount that potentially adverse consequences are limited in scope. The inevitable complexity of future cyber-physical systems (CPS) software will make it impossible to statically verify complete correctness. Formal methods, model checking and exhaustive testing of all possible control paths will either not be possible, or will not reveal all reachable system states (e.g., due to side-effects of cache activity or asynchronous events as a consequence of interrupts during run-time). A system should therefore be designed from the ground up to be resilient to unexpected control flow patterns and resource interactions. Central to our argument is that existing commercial off-the-shelf (COTS) systems have limitations in their designs that require a complete rethink in how they should be organized to support complex CPS software. A new approach to system design is necessary, to address challenges both in terms of software dependability and predictability.

For a system to be truly dependable it must guarantee that any unexpected behaviors, or software faults, do not manifest in a way that violates contracts between service providers and users. Here, a service user may be an application issuing a request to a system via an API that allows both functional and temporal constraints to be specified. Given a service request of this nature, it is the duty of the provider (here, the system) to ensure the application receives the desired service, even when unexpected situations arise. Such unexpected behaviors could be in terms of memory, CPU, and/or I/O protection violations. Examples include stray pointer dereferences, infinite loops, starvation, deadlocks, or uncontrolled access to I/O devices. Asynchronous control flow (e.g., due to interrupts, signal handling mechanisms, and multiple threads of execution), as well as resource sharing can lead to violations in performance guarantees and, hence, predictability. Other architectural factors, such as cache sharing, hyper-threading, and TLB design can affect the execution rates and, therefore, interference patterns between unrelated threads. Although static analysis techniques have been developed (e.g., in the design of type-safe languages) to address many forms of protection issues, this position paper suggests that additional run-time system support is necessary if software is to be made truly dependable.

We focus on two major OS challenges that need to be addressed in order to provide high-

confidence computing platforms that are resilient to inevitable unpredictable behavior. Specifically, we propose that (i) time be treated as a first-class entity, not only in the specification of task execution properties, but in the API's of the system itself, and (ii) the memory protection structure of the system be adaptable so as to maximize fault isolation (which adds execution overhead), while meeting application and system timing constraints.

2 Time as a First-Class System Property

The treatment of time as a first-class resource has been severely neglected even in operating systems that claim to be RTOSes. For the most part, these tend to provide static priority preemptive thread scheduling policies (via which rate-monotonic scheduling [4] and analysis can be performed), priority inheritance [6] for synchronization objects and critical sections, limited or no virtual memory/paging, and off-line profiling/worst-case execution time (WCET) analysis tools. Even relatively simple time-based scheduling policies such as "earliest-deadline-first" (EDF) have not wholly replaced static priority policies due to the added cost of managing dynamic priorities, and the potentially negative impact on task deadlines in unexpected overload situations. Even the POSIX.4 standard for real-time computing specifies `SCHED_FIFO` and `SCHED_RR` for "first-in-first-out" (non-preemptive) and "round-robin" (preemptive) scheduling of tasks with fixed priorities. Nowhere is there any specification of timing requirements in the API to the underlying system scheduler. Furthermore, the added costs of system services (e.g., as a result of scheduling and context-switching overheads, blocking delays due to synchronization on shared resources, and the impacts of interrupts from I/O devices) are not properly accounted in the timely execution of real-time tasks. It is no wonder, then, that systems designed for real-time computing still have significant areas of unpredictability. As further evidence, it is common for RTOSes to offer non-real-time network services via protocols such as TCP/IP, for compatibility with pre-existing applications rather than to ensure timing guarantees.

To better manage time so that predictable service execution is ensured, we propose to design a system interface that explicitly captures temporal constraints. That is, any requests from applications and/or higher-level services for a designated lower-level service will specify their timing requirements. Throughout the system, this timing information can be used to detect and possibly correct for deviant execution (e.g. due to unintended priority inversion). The inherent unpredictability of asynchronous events such as interrupts should be addressed by defining a unified scheduling hierarchy on every form of system "event". Specifically, scheduling decision points need to be placed at all locations in the system where control flow changes may impact the timing guarantees of tasks and their corresponding services. For example, one could devise a system that schedules interrupts in accordance with the priorities and timing requirements of tasks affected by those interrupts, rather than having interrupts always preempt potentially more important tasks. New device driver interfaces are needed to determine as early as possible which task is waiting on a given I/O response. When a device interrupt arises, it is essential that the urgency/importance of handling that interrupt is matched accordingly with the waiting process. Similarly, interrupts that are not associated with specific processes (e.g., inter-processor interrupts for TLB management, and clock interrupts to update system time) need to be correctly accounted in the timing requirements of tasks. None of these issues are adequately addressed by existing systems supporting existing APIs (e.g., POSIX). Only when time is treated as a first-class entity can true performance isolation guarantees be made.

3 Mutable Protection Domains

To limit the scope of adverse side-effects caused by errant or untrusted software, it makes sense to leverage, where appropriate, hardware and software-based memory fault isolation (i.e., protection) mechanisms [5, 3, 2, 1], thus allowing more predictable and controlled fault recovery. However, fault isolation overheads (e.g., due to page-table management, or run-time software safety checks) impact the granularity at which they can be imposed. They in turn impact the predictability of software execution, because factors such as TLB/cache misses, page replacement policies, garbage collection, and memory-bounds checks impose variable costs.

Fault isolation provisions of modern systems (e.g., Linux) are typically limited to coarse-grained entities, such as user- versus kernel-level protection domains and page-based process address spaces. μ-kernels [3] provide a minimal set of trusted services upon which higher-level services can be implemented at user-level. Both applications and user-level services typically map to separate processes which communicate via the kernel if they need to interact. Consequently, μ-kernels provide finer-grained fault isolation than monolithic systems at the cost of increased communication overheads. Virtual machines [1] allow multiple legacy OSs to be isolated from each other, while being able to co-exist on the same physical machine. Such a structuring provides very coarse-grained isolation boundaries, yet can impose significant communication overheads between VMs, and between VMs and the trusted virtual machine monitor. Regardless of the organization of the operating system, software-based fault isolation techniques can be employed to intercept references to invalid memory locations [5]. Similarly, type-safe languages can be used to detect potential software faults at compile- and run-time. Common to all techniques for fault isolation is the notion of a fault domain, such that one domain is isolated to some degree from another distinct domain, and an appropriately chosen method of communication is necessary for inter-domain communication. Significantly, all existing systems impose a static system structure, that is largely inflexible to changes in the granularity at which fault isolation can be applied. In turn, the method of inter-domain communication is static (e.g., a software trap for system calls, or an IPC message technique for inter-process communication). The efficiency and predictability of the system, then, is limited by the amount of overhead due to communication between isolation domains.

For the purposes of ensuring behavioral correctness of a complex software system, it is desirable to provide fault isolation techniques at the smallest granularity possible, while still ensuring predictable software execution. For example, while it may be desirable to assign the functional components of various system services to separate protection domains, the communication costs may be prohibitive in a real-time setting. That is, the costs of marshaling and unmarshaling message exchanges between component services, the scheduling and dispatching of separate address spaces and the impacts on cache hierarchies (amongst other overheads) may be unacceptable in situations where deadlines must be met. Conversely, multiple component services mapped to a single protection domain experience minimal communication overheads but lose the benefits of isolation from one another.

Given the above, we propose the design of a system with "mutable protection domains" (MPDs), that is flexible in its placement of fault isolation boundaries around various application and system components. Where possible, we attempt to maximize fault isolation, by mapping fine-grained software components to separate hardware protection domains, at the expense of increased communication overheads. In situations where such fine-grained isolation violates the acceptable end-to-end communication costs through a series of component services that are required to meet specific deadlines, we strategically increase the isolation granularity. The system, then, must decide where fault isolation boundaries are placed, using a combination of isolation benefit values applied to

the boundaries between software components, and the communication costs between components. The objective essentially involves the run-time adaptation of a system configuration to maximize fault isolation benefit while guaranteeing task timeliness constraints. Such benefit would ordinarily be gauged in terms of the otherwise adverse consequences that may arise if the corresponding level of isolation did not exist.

4 Summary

Given the increasing complexity of software systems, it will be almost impossible to statically verify their correct behavior. To ensure software dependability and predictability, existing COTS systems are deficient in several key areas that necessitate a rethink in their design. We propose a system design that considers two key factors for predictability and dependability. First, time should be treated as a first-class entity, requiring a revised API specification and a series of scheduling decision points to be placed at all locations where control flow changes may occur. Second, a flexible system structure that adapts protection domain boundaries and communication costs should be considered, so as to maximize fault isolation where possible while still ensuring predictability/timing guarantees.

5. Brief Biographies

Richard West received an MEng (1991) from the University of Newcastle-upon-Tyne, England, as well as both MS (1998) and PhD (2000) degrees in computer science from the Georgia Institute of Technology. He is currently an associate professor in the Computer Science Department at Boston University, where his research interests include operating systems, real-time systems, distributed computing and QoS management. Gabriel Parmer is a PhD student at Boston University, working on topics related to operating systems (especially their structure, service composition and extensibility), real-time systems and resource management. He currently holds a BA degree (2003) from Boston University.

References

[1] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[2] M. Fhndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys*, pages 177–190, April 2006.

[3] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[4] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 1973.

[5] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.