RESEARCH STATEMENT AND CONTRIBUTIONS

Gabriel Parmer
gparmer@gwu.edu

Most processors in the world today – over 97% – are embedded processors. Their software, which is responsible for controlling the likes of aircraft, medical devices and consumer electronics, is growing in complexity as users demand ever more functionality while at the same time requiring high reliability, real-time predictability, multicore efficiency, and manageable complexity.

Over the past six years, my research has focused on Operating System design and implementation, with the goal of building a modular, configurable, and fault-tolerant system. I have demonstrated that an embedded operating system – the COMPOSITE operating system developed by my students and myself – can be effectively and efficiently built from user-space, memory-isolated, fine-grained components. The purpose of this design has been to ensure predictable end-to-end execution, to isolate faults, to enable the pervasive configuration of resource management policies, and to harness the parallelism in modern systems. In real-time systems, predictability is particularly important and denotes the ability to place bounds on system execution, thus enabling its integration into a analyzable temporal schedule. Predictability is particularly difficult to achieve as it is a cross-cutting concern, thus dependent on large amounts of complex system code. COMPOSITE has demonstrated the ability to provide predictable execution for a component-based system within problem domains that are particularly important for current and future embedded systems.

In the description below, I provide a high-level overview of the key contributions, grouped around two themes: (1) Expanding predictability to new domains, in particular multicore and reliability; (2) Exploring the component-based paradigm for operating systems. Note: this condensed narrative refers only to some of the major papers; please refer to the CV for the remainder. COMPOSITE is open-source; see composite.seas.gwu.edu for details, and source code (> 140K lines of code).

1 Expanding Predictability Guarantees to New Domains

Traditionally, the study of predictability in embedded systems has often been confined to jobs and task scheduling, and the core functionality of a single processor OS. In our work, we have investigated means to more effectively harness parallelism, and recover from faults within system-level components, while *also* ensuring that such systems adhere to analyzable timing bounds.

FJOS: predictable, efficient fork-join parallelism. To continue increasing the functionality and capabilities of future embedded systems, it is necessary to harness the parallelism provided by current and future multi-core hardware. Toward this, we have developed FJOS – the fork-join OS – as a set of low-level components on top of COMPOSITE to enable tasks to harness the massive parallelism of multi-core systems. FJOS supports OPENMP, a high-level, prominent language extension (including support in C) for fork-join parallelism. At its heart, the FJOS implementation features data-structures and abstractions that effectively exploit cache-coherency for inter-core communication, and low-level, controlled access to inter-processor interrupts (IPIs) for inter-core thread activation. To effectively analyze the timing of the system *and* efficiently use the hardware, we also provide a task assignment algorithm (of parallel threads to cores) for fork-join task sets, and an analysis to assess schedulability.

FJOS has demonstrated its utility relative to Linux by better utilizing the parallelism available on a current 40 core system (with between 2.5 and 15 times less overhead than Linux). Notably, blocking implementations of OPENMP synchronization in FJOS are on par with spinning implementations in Linux, thus enabling the co-execution of multiple OPENMP tasks, and the more effective power-management of such systems. This work yielded [1], which won the *best student paper award* at RTAS '14.

Future research in predictable parallelism. FJOS is the first in a series of contributions that effectively and predictably harness parallelism to continue and empower relentless increases in embedded system functionality. More broadly, we are currently working on the design of a system with *scalable predictability* – a system with the ability to maintain a given bounds on execution, independent of the number of cores in the system. This is particularly important in mixed-criticality systems in which software of various quality levels must be temporally and spatially isolated. This goal requires a complete rethinking of the kernel data-structures and organization, but will yield a firm foundation for predictability in future massively multi- and many-core systems.

C^3: predictable, system-level fault tolerance. As process technologies continue to shrink transistors, thus shrinking chip footprint and increasing computational power, a significant danger is an increase in the number of faults due to hardware defects, aging, and external effects (e.g. cosmic rays). In fact, to surmount these challenges, a call for a "concerted effort on the part of all the players in a system design" was made [2]. We have focused on a particularly pernicious family of faults: those that impact core software in the OS such as the scheduler, memory manager, and file-system. A simple software or hardware fault in any of these critical components easily results in complete system failure. The challenge in tolerating faults in such components is not only to avoid broad corruption of system state and functionally recover the system, but *also* to do so in such a manner that recovery is bounded in execution (predictable), while also doing so efficiently enough to not disrupt the system's main job (e.g. physical system control).

To address the challenge of embedded system fault tolerance, we have designed and implemented the Computational Crash Cart (C^3): a set of mechanisms implemented on COMPOSITE to not only recover from system-level faults, but to do so within a bounded, predictable amount of time. C^3 uses a combination of a novel interface-driven recovery, and efficient, predictable micro-reboot to recover the state of failed system components. Micro-rebooting a failed component enables recovery of a "safe state", but leaves the component inconsistent with other components around it. A consistent state is recovered using the very functions exported by the failed component to rebuild the objects (e.g. threads, files) expected by the rest of the system. We have studied not only component recovery times (which are small), but also their impact on system timing. When compared with an efficient embedded implementation of checkpointing, C^3 is significantly more effective at system recovery for all but the smallest (memory) systems [3]. In comparison, existing mechanisms (e.g. checkpointing in Xen and Linux) have an order of magnitude higher recovery overhead and are not predictable. This work is funded by an *NSF CAREER award* and has yielded multiple publications including [3].

Future research in predictable fault tolerance. C^3 is a foundation on which to build a truly fault-tolerant system. However, the vision of a system that can recover from all types of faults (e.g. latent, deterministic) requires additional mechanisms that can comparably provide efficient and predictable guarantees. For example, we're investigating a pervasive monitoring infrastructure to log all component interactions to predictably detect faults that don't immediately trigger an exception. All such future directions harness the flexibility, interfaces, and inherent fault containment provided by the unique model and implementation of COMPOSITE.

2 Exploring the Component-Based Paradigm for Predictable OSs

Traditional monolithic operating systems have grown to the point where it is difficult to harness them to address increasingly important challenges in embedded systems in the areas of system reliability, security, and maintainability. One of my main research goals has been to explore the design and implementation of a predictable component-based operating system to serve as the foundation for future embedded systems. This section gives a brief overview of the contributions therein.

Component-based, user-level, configurable scheduling. Many modern μ-kernels are driven by the philosophy, "a concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the systems required functionality" [4]. Since the advent of L4 and other second (and third) generation μ-kernels, which adhere to this philosophy, one functionality of crucial importance remains in the kernel: *scheduling and time management*. Moving scheduling policy to user-level components is particularly difficult as it has implications on interrupt handling, and IPC. In fact, a recent paper [5] emphasized the difficulty of and lack of solutions in providing user-level scheduling in L4. In spite of this, COMPOSITE provides efficient and predictable user-level, component-based scheduling through the fundamental abstractions and mechanisms provided by the kernel. We avoid scheduler invocation on the IPC path – the path that *must* be fast – by adopting a migrating thread model [6], and COMPOSITE provides efficient interrupt execution scheduling via a kernel/scheduler communication channel. Using these facilities to configure and protect the timing properties of the system as a user-level component, we've demonstrated the ability to achieve a difficult combination: *both user-level, configurable scheduling,* and *efficient, predictable system IPC and interrupt delivery* [7, 8]. This user-level scheduling foundation has been integral in enabling many other facets of our research including FJOS and C^3.

HIRES: hierarchical resource management and virtualization. Though user-level scheduling enables an embedded system to customize the scheduling policies it uses, it is necessary in some systems to enable different subsystems to each customize their own resource management policies. As embedded systems are required to go beyond the traditional bounds of scheduling simple, trusted computation to managing multiple complex subsystems of varying levels of criticality and trust, the ability to isolate and customize each subsystem is invaluable. Toward this, HIRES enables the *arbitrary nesting of subsystems* by virtualizing resource management for CPU, I/O, and memory. Each subsystem uses appropriately-defined management policies, and HIRES provides protocols and mechanisms to enable the *delegation* of resource management between these subsystems. In contrast to traditional virtualization infrastructures, the mechanisms in HIRES enable a constant and predictable cost for managing all resources, regardless of a subsystems place in the hierarchy. Using HIRES we've paravirtualized FREERTOS, and in the future will expand our virtualization support to include Linux. HIRES has been integral in enabling specialized subsystems (e.g. supporting C^3 and FJOS) to exist alongside traditional or best-effort software environments. HIRES has yielded a publication [9] that was *nominated for the best-paper award* at RTAS.

Future research in assurance for decentralized OS resource management. Just as access control frameworks are used in security-focused systems to control the flow of information throughout a system, independent of the computations that act on it, we are in the process of evaluating TCAPS, or temporal capabilities, a system for doing the same for the *flow of time*. As virtual machines and mixed-criticality systems encourage the decentralization of resource management between untrusting subsystems, TCAPS make guarantees on the processing allocation between subsystems that are made *independent* of the actual resource management policies that might be buggy, or compromised. Building on user-level, component-based scheduling and HIRES, TCAPS control the delegations between subsystems, and the "quality" of the time passed (e.g. what priority it has). TCAPS represent a complete rethinking of the infrastructure for time management, and is our first push into the security domain.

Predictable end-to-end component-based execution. Many systems have the capability to break software into separate, isolated pieces, for example using processes in Linux, or servers in μ-kernels. Existing solutions commonly use communication between separate threads (synchronous or asynchronous). Such systems make end-to-end timing analysis more difficult and generally pessimistic by explicitly requiring the consideration of task dependencies in the scheduling analysis. This raises the fundamental question: for predictable systems that can benefit from inter-process communication (or IPC), what is the best execution model for system tasks? Answering this question with a system that is efficient and predictable is essential to the success of a component-based OS, which has highly frequent IPC.

COMPOSITE uses a combination of the migrating thread model for IPC, and a system we call *transient memory* (TMEM) to provide end-to-end predictable and efficient system execution. TMEM solves a simple problem: what memory should a thread use as an execution stack when it enters onto (i.e. invokes a function in) a component? TMEM provides a set of protocols and mechanisms to manage pools of memory to be used for this purpose, and transforms the traditional dependent thread problem into a resource sharing problem on TMEM pools. Traditional protocols are used to mediate this sharing, and the system optimizes the size of each component's pool to directly manage the end-to-end latencies of different threads. Our work on transient memory has yielded two publications that cover both stack and shared memory management for both hard and soft real-time systems [10, 11]. TMEM represents a significant interplay between theory and implementation by providing a novel system structure that reframes the system model into an existing theoretical problem, and then defines an optimization problem (assigning memory to component pools) on top that considers and optimizes for implementation overheads.

Mutable Protection Domains (MPD). A limitation of component-based systems that utilize hardware memory isolation functionality is that communication between components implies some overhead. However, the flexibility and reliability of the system requires small, fine-grained components that imply more communication. With MPD, we study the ability to *dynamically* raise or lower protection boundaries between components, thus enabling the system to consciously trade-off isolation for overhead [12, 13]. We show that for a specific application (web serving), the overhead "hot spots" change depending on the workload, thus motivating MPD's dynamic capabilities. Using MPD, we show that 40% of IPC overheads can be removed while maintaining over 80% of the protection boundaries in a complex system. This motivates the development and use of fine-grained components, rather than the common practice of manually creating coarser components to "avoid overhead".

3 Summary and Future Research Directions

Our research has two main themes: (1) we build efficient mechanisms for operating systems – fault tolerance in C^3 and inter-core synchronization and activation in FJOS – that have significant utility on their own, but also provide *predictablity, therefore real-time guarantees*; and (2) we make contributions in the field of real-time, component-based operating systems by demonstrating their practicality (end-to-end predictability, TMEM, and MPD), and by investigating their unique capabilities (user-level, component-based scheduling and HIRES). COMPOSITE provides a unique infrastructure to continue to investigate future embedded systems that have stringent requirements for pervasive configuration and isolation, massive parallelism, dependable execution, and heightened security. COMPOSITE also has a great promise to "scale up" effectively from embedded systems to data-centers and HPC where concerns about predictability are increasingly prominent. Given COMPOSITE's flexibility, we believe it will be a significant vehicle for continued contributions for many years to come.

References

[1] W. Qi and G. Parmer, "FJOS: Practical, predictable, and efficient system support for fork/join parallelism," in *accepted in the Proceedings of the 2014 20th IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, 2014.

[2] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, 2005.

[3] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *Proceedings of the 2013 34th IEEE Real-Time Systems Symposium (RTSS)*, pp. 21–32, 2013.

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

[5] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 133–150, 2013.

[6] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2010.

[7] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[8] G. Parmer and R. West, "Predictable and configurable component-based scheduling in the composite os," *ACM Transactions on Embedded Computer Systems*, vol. 13, pp. 32:1–32:26, Dec. 2013.

[9] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[10] Q. Wang, J. Song, G. Parmer, G. Venkataramani, and A. Sweeney, "Increasing memory utilization with transient memory scheduling," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, 2012.

[11] Q. Wang, J. Song, and G. Parmer, "Stack management for hard real-time computation in a component-based OS," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[12] G. Parmer and R. West, "Mutable Protection Domains: Towards a component-based system for dependable and predictable computing," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, pp. 365–378, 2007.

[13] G. Parmer and R. West, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *ACM Transactions on Software Engineering (TSE)*, July/August 2012.