Motivation and Vision

My research addresses two areas of experimental systems. First, I am developing techniques and tools to support the design, optimization, and validation of software for embedded systems. Second, I am striving to create operating systems that support multimedia and other real-time applications by coordinating resource allocation and user preferences, making use of CPU scheduling techniques, and managing scheduling latency and other forms of timing unpredictability.

The vast majority of new microprocessors are used in embedded systems: computers that directly interact with the physical world, whose presence is often non-obvious to casual users. My primary long-term goal is to make it easier to rapidly develop robust embedded software that requires minimal hardware resources. I do not believe in a single “silver bullet” that will accomplish this goal. Rather, over the next few years I plan to attack problems in developing embedded software from a variety of angles. For instance, I am

• taming concurrency problems by creating programming environments tailored to particular sub-problems of the overall embedded system design problem;

• designing and building tools that analyze and optimize embedded software without requiring the developer to give up control over low-level implementation details;

• developing new programming models and abstractions that encapsulate currently-problematic program properties such as security and timing predictability.

This research agenda will yield a body of techniques and tools that permit skilled but non-expert systems designers to quickly create efficient software for a new embedded system or to make meaningful changes to an existing embedded system. Furthermore, developers using these techniques will have high confidence that the system will operate correctly, since many classes of errors will be detected statically.

My secondary research agenda helps create consumer operating systems that permit multiple, independently developed soft real-time applications to run concurrently, and that also accurately ascertain end-users’ resource allocation preferences and respect them when there is contention. Supporting soft real-time applications is important not only because of multimedia, but also because of the increasing popularity of low-cost peripherals that rely heavily on the main CPU for real-time processing. My research in this area will focus on systems that are resilient to changes in application requirements and system capabilities, and on making application development easier, e.g., by relying on the system, rather than the programmer, to determine application resource requirements.

My research efforts are tied together by a few common themes. First, I believe that good systems research is problem-driven rather than solution-driven. Consequently, I take a flexible approach that draws on techniques from diverse areas of computer science. Second, it is important to establish an appropriate balance between theoretical research and implementation; I find both types of work indispensable. Finally, much of my research parallels some general trends in systems research. There is increased desire for automated formal reasoning about properties of systems software, and for language-based support for programming abstractions that were traditionally implemented idiomatically. Also, there is a growing focus on domain-specific extensibility in service of specific classes of applications, rather than supporting general-purpose extensibility in systems.
Building Embedded Systems with Composable Execution Environments

My vision for embedded systems centers on Composable Execution Environments (CEE). I am the principal author of a grant proposal on this topic that is funded by NSF in Embedded and Hybrid Systems (award CCR-0209185). CEE is based on two main ideas. First, embedded systems are best structured as a collection of restricted execution environments that facilitate testing, debugging, and verification by restricting design freedom along certain axes. In contrast, unrestricted designs, such as those typically produced by non-expert programmers using preemptive multithreading, can be very difficult to analyze or understand. Second, execution environments can be composed hierarchically using appropriate analyses to ensure the safety of compositions. My current research efforts, outlined below, are fleshing out components of CEE.

Concurrency Analysis. With Alastair Reid I developed task/scheduler logic (TSL) to reason about concurrency in systems software where properties of execution environments are propagated hierarchically [11]. This is a very common pattern; for example, a non-preemptive event loop run in interrupt context inherits interrupt properties such as a very high priority and inability to block. TSL usefully exploits these inherited properties and we have used it to determine the presence or absence of race conditions in real systems. TSL can also be used to identify redundant synchronization calls and to infer which of the available synchronization primitives satisfy the atomicity requirements of a particular critical section.

Robust Real-Time Scheduling. A serious problem with existing theories of real-time scheduling is that they usually assume prior knowledge of worst-case execution time (WCET). Computing WCET is difficult at best, and even when a bound on run-time is available it may be too pessimistic to be useful in practice. This led me to develop robust scheduling: a systematic way to create a schedule for a real-time system that maximizes its robustness with respect to timing faults — tasks that run longer than their nominal WCET [6].

Mixing Preemptive and Non-Preemptive Scheduling. Most existing models for real-time scheduling deal with either fully preemptive or fully non-preemptive scheduling. To support CEE I developed task clusters and task barriers, programming abstractions that give developers access to a fairly general mix of preemptive and non-preemptive scheduling [6]. I also developed algorithms for scheduling these new abstractions. It is important to mix preemptive and non-preemptive scheduling because non-preemptive scheduling has large software engineering benefits — many kinds of race conditions and deadlocks become impossible — while in many cases some preemption is still required in order to meet response time requirements.

Bounding and Minimizing Stack Size. Many small embedded systems must fit into on-chip memory, often sized around 512 B–4 KB. An important element of overall system safety, that is not addressed by traditional type-safe languages, is stack safety: a static guarantee that a program will not overflow its execution stack. I have recently developed a novel method for bounding the worst-case stack depth of embedded programs; it handles complications such as indirect jumps and nested interrupt service routines. I am also using the stack size analysis as a “subroutine” in a more ambitious effort to automatically reduce the stack memory requirements of embedded programs.

In the near future I will continue to realize CEE. I also plan to create an embedded testbed: a room full of devices such as small robots, PDAs, and networked sensor nodes. Bringing together a collection of diverse gadgets will provide a rich environment for students to learn about embedded systems, and since good systems research is often motivated by practical experience it will help spur new research projects. In the longer term I will work on ways to create distributed embedded systems that have desirable end-to-end properties. This is difficult, in part, because our current abstractions are too narrowly conceived: they focus on individual nodes and connections, with useful high-level properties “emerging” unpredictably. Understanding, predicting, and controlling these emergent properties are important research challenges.
System Support for Real-Time Applications

Operating system support for soft real-time applications has become well understood during the last five years. However, challenging problems remain, such as coping with highly variable resource availability on a modern power-conscious machine. This section describes my past research in this area and identifies areas for future work.

**Composing Real-Time Schedulers.** My dissertation work is on hierarchical loadable schedulers (HLS), a framework for composing real-time schedulers in the kernel of a general-purpose operating system [4, 10]. Hierarchical scheduling makes it easier to support concurrently running applications with diverse requirements and to recursively apply scheduling properties, such as real-time guarantees, to groups of threads. The primary contribution of HLS is a framework for reasoning about the guarantees provided by a heterogeneous hierarchy of schedulers. To demonstrate its efficiency and its effectiveness in solving scheduling problems, I implemented HLS in the Windows 2000 kernel.

**Programming Models for Multimedia.** Many multimedia scheduling techniques require developers to supply a priori knowledge about applications: this complicates application development and is a significant obstacle to the widespread adoption of these methods. I evaluated a number of multimedia scheduling techniques in the context of the programming models they present to developers, and the usage models they present to users [7]. This work is useful because it proposes new metrics for evaluating multimedia support, and these metrics capture important real-world effects such as developer effort.

**Middleware Resource Management.** Although there exist a number of technologies for real-time CPU scheduling in general-purpose operating systems there is often no easy way for programs — and legacy programs in particular — to make use of these facilities. I have designed a CPU Resource Manager [8] that can monitor (and in some cases, infer) application resource requirements and interact both with users and with the underlying operating system to ensure that resources are allocated in accordance with high-level policies.

**Inferring Scheduling Behavior.** Validating CPU schedulers and other forms of system support for real-time applications is difficult. To address this, I created Hourglass [5], a tool for making inferences about scheduling behavior purely from observations made at user level. For example, it can be used to measure both indirect (cache-related) and direct costs of context switches and to model applications with hard real-time deadlines, reporting precisely which deadlines they hit and miss. Hourglass is in use at several universities and one company; feedback about it has been positive.

**Managing Latency and Unscheduled Time.** Regardless of the high-level strategy that an operating system uses to manage resources, its ability to support real-time applications can be limited by scheduling latencies. Mike Jones and I investigated sources of latency in Windows 2000, and fixed some of them [2, 3]. I developed augmented CPU reservations, a lightweight mechanism for preventing an OS from stealing cycles from time-sensitive applications [9].

**Scheduling Abstractions for Soft Real-Time.** CPU reservations and time constraints are abstractions designed to support coexisting soft real-time applications. Together with Mike Jones, I designed and implemented Rialto/NT, a system that brings the benefits of these abstractions to Windows 2000 users [1].

In the future I will focus on inferring application requirements and otherwise making developers’ jobs easier, designing systems whose performance is robust with respect to uncertainty in workload and system capabilities, and attempting to create systems that do what users want rather than relying on them to make manual adjustments.
References


