

This document summarizes the chief activities and findings for “Composable Execution Environments: A Foundation for Building Robust Embedded Systems,” supported by National Science Foundation Award CCR-0209185, over the project’s entire duration (2002–2005). Jay Lepreau was the PI, and most of the work was managed by co-PI John Regehr.

Overview

Our work on this project focused on solving problems created by the tension between the need to develop embedded software as a collection of components, and the fact that the overall acceptability of an embedded system is a multi-faceted global problem. Global system acceptability issues include:

- ensuring freedom from concurrency errors,
- meeting real-time deadlines,
- meeting throughput requirements,
- staying within RAM and ROM limits, and
- obeying system-specific rules (no blocking in interrupt mode, etc.).

The basic technical approach of the CEE project was to combine ideas from systems and languages in order to solve these problems. In particular, we developed tools supporting rapid creation of high-quality embedded software from components. The target platform for essentially all of our research has been sensor network nodes: the Berkeley mica2 motes running TinyOS.

Composable Preemptive and Non-Preemptive Real-Time Scheduling

Our work required a model for scheduling real-time tasks that extended the state of the art, supporting a flexible mix of preemptive and non-preemptive scheduling. Mixing these two kinds of scheduling is useful because preemption permits flexible allocation of CPU time while non-preemptive scheduling has many benefits in software engineering (since race conditions become much more difficult to create) and efficiency. To support this model we developed two new abstractions, *task clusters* and *task barriers* that are described in our paper from RTSS 2002 [3].

Limitations of current techniques for measurement- or analysis-based estimation of worst-case execution time (WCET) caused us to develop a way of evaluating real-time systems for their robustness to timing faults, or tasks that run for longer than expected. In our RTSS paper [3] we also described some theoretical results about the robustness of real-time schedules produced by existing scheduling algorithms, and we developed algorithms for creating highly robust schedules that are as resistant as possible to timing faults.

Finally, as part of this work we developed and released the *static priority analysis kit* (SPAK) that provides an open-source implementation of all algorithms described in our paper from RTSS 2002.

Eliminating Component Overhead

Run-time overhead is often perceived to be a drawback of component-based software. We were interested in techniques that can be used to reduce or eliminate the performance overhead of components. To this end we extended an existing tool developed by the Flux group, the *flattener*, a

cross-module inlining tool, in order to permit it to accept an externally-specified list of inlining decisions. We have successfully increased the performance of some large programs using the flattener, and we released it as open-source software during Summer 2003.

Lock Inference

A major problem in developing embedded software is that these systems are concurrent, leading to race conditions that are hard to find and fix. We developed the task scheduler logic (TSL), a novel way to reason about concurrency in systems software. The primary contribution of TSL is that it permits concurrency analysis of programs even when they contain multiple execution environments, such as preemptively scheduled interrupts, preemptive threads, non-preemptive events, and non-preemptive dataflow elements. We have also developed techniques for performing *lock inference*—the inference of an appropriate lock implementation for each critical section in a system [6]. Lock inference is important because it facilitates composition and reuse by reducing the number of assumptions that software components have to make about their execution environments.

Inferring Real-Time Scheduling Behavior

When developing real-time and embedded systems, it is often desirable to modify or tune CPU scheduling algorithms, and even when schedulers are not modified it is often useful to study their behavior to ensure that it is correct and predictable. We developed a user-level program called *Hourglass* that provides tool support for making inferences about the behavior of schedulers. Hourglass supports the measurement of dispatch latency, context switch overhead, and the ability of tasks to meet real-time deadlines under a wide variety of conditions. Hourglass was the subject of a research paper [2]. It is available as open source software.

Creating Evolvable Embedded Systems

In a paper published at RTSS 2003 [9] we described the results of an experiment in creating real-time embedded software that is *evolvable*: it is resilient to the changes in requirements and structure that inevitably occur over time. The primary finding of this work is that a hierarchical concurrency analysis and a real-time schedulability analysis can be combined to create a new way to look at embedded software: as a collection of execution environments, each of which has unique properties that make it a useful place to run code. By making execution environments explicit (we did not invent them, but rather identified them as an important, previously neglected, aspect of embedded software creation), and by making it easy to move code between them, we showed that highly nontrivial changes to the structure of embedded software can be made quite easily.

Bounding and Reducing Stack Depth

We presented a paper at EMSOFT 2003 [8] about bounding and reducing stack depth for embedded software. The primary contributions of this work were (1) to show that it is possible to tightly bound the worst-case stack memory requirements of embedded programs compiled from up to about 30,000 lines of C, and (2) to show that stack depth can be effectively reduced using goal-directed whole-program inlining. As far as we know, this was the first paper on automatically reducing stack depth. Saving memory is important on small, cheap devices like the Berkeley motes, our target platform, as they typically have 4 KB or less of RAM. Although TinyOS, the software system for the Berkeley motes, is component-based, our conclusion was that a compositional approach to reasoning about stack depth was unlikely to work well. In other words, since the

compiler deliberately blurs lines between components at compile time to eliminate cross-component overhead, a brute-force whole-program analysis is preferable.

We have continued development on our stack depth analyzer and we have also released it as open-source software on our web site. We also investigated a second method for reducing stack memory requirements: eliminating unnecessary preemption relations among interrupt handlers. In a paper that has been accepted to be published in ACM Transactions on Embedded Systems we report the results of this technique, which reduces worst-case stack depth of TinyOS programs by up to 28%. Finally, we wrote an article about bounding stack depth that was published in a trade magazine that is widely read by embedded system developers [4].

Constructing Static Analyzers

While working on our stack depth analysis tool, we found that creating an abstract interpreter for machine code was a tedious and error-prone task. This difficulty motivated a new project, Hoist, which automates the construction of the abstract transfer functions that form the bottom layer of an abstract interpreter. This work is described in a paper that was published in ASPLOS 2004 [7].

Hoist uses an embedded CPU as its own specification in order to learn the behavior of its instruction set. This concrete behavior is then lifted into an abstract domain (currently we support the well-known interval domain and also the “bitwise” domain used in our stack analysis work) and the abstract transfer function is efficiently encoded using a BDD, which is then used to generate C code that is ready to be linked into an abstract interpreter. Hoist is highly generic, supporting multiple abstract domains and, we believe, multiple architectures (currently we support only the Atmel AVR).

Hoist works well for small embedded architecture but it does not scale to, for example, 32-bit architectures. Recent work by masters student Usit Duongsaa [1] has shown that it is possible to derive maximally precise abstract transfer functions for the interval and bitwise domains such that the derived operations run in time linear in the bitwidth.

Protecting Against Interrupt Overload

Composable software relies heavily on isolation mechanisms. For example, reservation-based schedulers permit real-time workloads of varying criticality to coexist on a single processor. Similarly, hardware- or language-based memory isolation permits secure and insecure code to share a single processor.

Existing isolation mechanisms fail to protect against *interrupt overload*, a condition that occurs when external interrupts are signaled at a high enough rate that processing is starved. This can occur despite the efforts of thread-based reservation schedulers. Interrupt overload is particularly a danger when an existing embedded system is connected to a network—an adversarial network node could flood the system with small-sized packets. For example, even 10 Mbps Ethernet can generate more than 14,000 interrupts per second.

We developed several “interrupt schedulers” to protect against interrupt overload. An interrupt scheduler enforces a maximum arrival rate for an interrupt source, dropping interrupts during overload. This protects other activities running on the processor from starvation. The schedulers provide different tradeoffs between overhead, ease of implementation, and granularity of protection. We showed that they are effective and efficient. A paper describing these results was published in LCTES 2005 [5].

Compiling for Non-Functional Constraints

Embedded systems are subject to diverse and sometimes severe *non-functional constraints*, for example real-time deadlines, throughput requirements, and finite amounts of different types of memory. Meeting these constraints by manual tuning of code is problematic for many reasons: it breaks modularity, inhibits reuse, introduces bugs, and is time consuming. Even so, existing compilers provide very little support for meeting non-functional constraints. We began, but have not completed, work on a framework for *application- and system-specific heuristics* to control low-level compiler optimizations in order to meet non-functional constraints [10]. Our initial results, applied to large TinyOS applications, are promising.

References

- [1] Usit Duongsaa. Automatic generation of abstract transfer functions. Master's thesis, University of Utah, July 2005.
- [2] John Regehr. Inferring scheduling behavior with Hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, Monterey, CA, June 2002.
- [3] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symp. (RTSS)*, Austin, TX, December 2002.
- [4] John Regehr. Say no to stack overflow. *Embedded Systems Programming*, 17(10), October 2004.
- [5] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.
- [6] John Regehr and Alastair Reid. Lock inference for systems software. In *Proc. of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Boston, MA, March 2003.
- [7] John Regehr and Alastair Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.
- [8] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.
- [9] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, December 2003.
- [10] Alastair Reid, Nathan Coopriider, and John Regehr. Compiling for resource-constrained platforms using ASSHes: Application- and system-specific heuristics, November 2005. Unpublished draft. <http://www.cs.utah.edu/~regehr/papers/assh-draft.pdf>.