Program Analysis and Implementation Techniques for Real-Time and Embedded Software Interoperability

Abstract

We propose a new set of program analysis and implementation technologies to enable the integration of mainstream development practices and languages into the development of real-time and embedded systems. These technologies include analyses that extract the resource requirements of the real-time components and a set of synchronization and scheduling techniques that make it possible for these two traditionally disjoint kinds of software to interoperate.

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinnard@lcs.mit.edu

October 30, 2001

1 Introduction

Real-time software and embedded software has a set of extreme requirements: the system must execute reliably and predictably, consume a limited and predictable amount of resources, and meet harsh real-time constraints for responding to events in the real world. These requirements have led to development strategies that implement real-time systems in isolation using custom hardware and software such as real-time operating systems. Programs typically are written in a constrained style to ensure predictable execution times and memory consumption.

In the future there will be enormous economic pressure to reintegrate the field of real-time and embedded software to use languages and development practices drawn from more mainstream approaches. These approaches tend to use a set of abstractions and concepts (dynamic memory allocation, garbage collection, arbitrary object references) that lead to good programmer productivity but conflict with the standard real-time system requirements of reliability, minimal resource use, and preditable execution.

We see two levels of integration as desirable. The first is for real-time and embedded software to interoperate successfully with other parts of the system (such as user interfaces) that were originally developed for other, less stringent environments. The second is to integrate development approaches and engineers from more mainstream areas of software development. In both cases, it will be extremely desirable to use mainstream languages as the implementation vehicle for the real-time and embedded part of the software.

2 Our Approach

We propose to develop technology to enable developers to use standard languages (such as Java and other dynamic object-oriented languages) to develop real-time and embedded systems. We see the key issues as 1) using program analysis to extract predictability guarantees for the real-time and embedded components of the overall system and 2) develolping implementation mechanisms that allow real-time and embbeded software to interoperate with parts of the system originally developed for other less-constrained purposes.

3 Static Analysis

Several areas of static analyisis will be crucial to the success of this endeavor:

*This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and an NSERC graduate scholarship.

- **Memory Consumption:** We expect to see the deployment of real-time and embedded software (written in Java, for example) that uses dynamic memory allocation. In this scenario, we see a need for program analysis tools that can compute the amount of memory required to execute each real-time or embedded computation. The system would then set aside that amount of memory to ensure that the computation would never require more resources than are available.

- **Object Lifetimes:** To eliminate garbage collection pauses in real-time software, we see a need for a program analysis that can analyze the lifetimes of objects and use an explicity allocate/free memory management strategy for real-time code.

- **Region-Based Memory Allocation:** Region-based memory allocation is a promising strategy for real-time systems because it enables the system to allocate a block of memory for a computation, run the computation without memory management overhead in that region, then deallocate the region as a unit when the computation terminates. This strategy is attractive enough to have been adopted in the recent Real-Time Specification for Java.

  The problem is that the use of such constructs raises the possibility of dangling references that go from the standard heap memory to the region memory. We see a need for a static analysis to verify that the program never creates these references. Such an analysis would enable the software to safely use region-based memory allocation.

- **Worst-Case Execution Time Analysis:** We see a need to analyze the program to compute worst-case execution times for real-time computations. We see the primary challenge as uncertainty about the memory system behavior caused by dynamic memory allocation and a potential lack of control over the addresses containing data.

4 Implementation Mechanisms

We see a variety of implementation mechanisms as required to enable real-time and embedded software to safely interact with software written for more mainstream applications. We assume that these two kinds of software will share objects and interact by performing atomic operations that update the shared objects.

In this context, mutual exclusion raises the possibility that the non-real-time software could unacceptably delay the real-time software. Such delays could come from garbage collection pauses during atomic operations (note that this problem eliminates priority inheritance as a solution to the problem of delaying

real-time threads) or from faulty mainstream applications that fail while holding locks or simply neglect to release the lock.

In this scenario, the software needs a way to make the atomic operations of the real-time system execute *regardless* of the actions of the other parts of the system. We see optimistic synchronization operations on multiple objects as the way to eliminate this problem. The challenge is to develop protocols that operate effectively and efficiently enough to be used in deployed systems.

A second implementation problem has to do with real-time scheduling. Traditional systems use a real-time operating system with real-time threads. This approach places a large amount of operating system code between the application and the realization of its real-time constraints. We instead propose a software-enabled scheduling mechanism in which, under the control of code automatically inserted by the compiler, the real-time threads continually check to see if they should activate a new task with real-time constraints. We expect the advantages to include faster response time to real-time requests and more control over the real-time scheduler. Ideally, engineers will be able to exploit this control to deliver useful scheduling algorithms that better control the interaction with the real world.

5 Conclusion

We propose a new set of program analysis and implementation technologies to enable the integration of mainstream development practices and languages into the development of real-time and embedded systems. These technologies include analyses that extract the resource requirements of the real-time components and a set of synchronization and scheduling techniques that make it possible for these two traditionally disjoint kinds of software to interoperate.