Better Abstractions: an Agenda for Embedded Systems Research

Andrew P. Black
black@cse.ogi.edu

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon, USA

Panel Presentation at DARPA-NSF Workshop on Embedded & Hybrid Systems

Introduction

For many years I have been part of the operating systems community; the focus of this presentation is to venture a "systems view" of some of the research that is needed in embedded systems.

What operating systems do is provide programmers with implementations of abstractions, and operating system researchers have been described as "Abstraction Merchants". The trick to being a successful abstraction merchant is to provide the functionality that your clients need, in a convenient form, and at low cost. It is also important that abstractions "don't hide power" (Lampson 1983), or at least that they don't hide the power that your users need to wield.

Embedded systems may be the one area of computing where successful abstractions are still lacking. In 1973, when I took my first job in the computer industry, one of my tasks was to write parts of an experimental operating system—in assembly language. No one writes operating systems, or compilers, or programming environments, in assembly language any more. But embedded control programs are still routinely written in assembly language, or in something at a similarly low level.

Why have 30 years of development in programming languages had little or no impact on embedded systems? I believe that the answer is that programming language designers, among whom I count myself, have built the wrong abstractions—for Embedded Systems.

It is in the nature of an abstraction that it emphasises some features of a system by abstracting away from, that is, ignoring, other details that are felt to be irrelevant for a particular purpose. One detail that all conventional programming languages have chosen to ignore is time. This happens to be a detail that few embedded systems can afford to ignore.

When programming language designers and implementors speak of “equivalent programs” or “alternative implementations”, they are abstracting away from time. The “equivalent” programs may be semantically equivalent, but they may differ widely in their execution efficiency and in their effects on other processes due to interference through the scheduler.

The Benefits of Abstraction

I believe that the next generation of embedded systems must be adaptive, in the sense that resources must be allocated and controlled dynamically, rather than statically. There are a number of reasons for this, starting with the increasing complexity of applications, which may render traditional static analysis impossible or impractical, and the effective non-determinism of the hardware. The wide availability of cheap, powerful processors means that economics is forcing us to put many applications on the same chip, where resources consumed by one application are not available for another. But perhaps the most compelling reason is that as embedded systems interact with the environment in more comprehensive ways, the effects of that environment, and of other interacting systems, become impossible to predict. Wireless networking is a good example: the bandwidth available to an individual application is affected not just by the applications sharing the processor and the transceiver, but by all the other “applications” sharing the same ether.

The key question when building an adaptive embedded system is: if we adapt the parameters and algorithms inside an application, how can we ensure that the applications’ behaviour remains stable and appropriate? In my view the answer to that question is that we must be explicit about the behaviour that we want, rather than having it emerge as the consequence of a thousand individual implementation decisions. That’s exactly what abstractions are for: making the important behaviour explicit.

An example of this struck me only a few days ago. A colleague said to me “this piece of code must execute without pre-emption”. Why? What was the abstraction that he was trying to capture? There are at least two possibilities. First, that the executing thread must be “back” within a given response time, because it is needed to service some external input signal or event. Or, second, that the code temporarily disrupts the invariants of a shared data structure, and prohibiting pre-emption meant that no other thread could observe that disruption. Whichever of these properties is important must be preserved when we compose small fragments of functionality into larger and more complicate applications. How can we know which of these properties the ban on pre-emption is trying to preserve? Indeed, how can we hope to preserve something that isn’t even written down?

Instead, I believe that we should provide abstractions that allow the programmer to state the inter-event latency of the input signals *explicitly*, and let the system take responsibility for having a thread available to handle them, or for informing us that this is impossible. Similarly, we should protect the manipulation of shared data structures with critical regions. The implementation of these regions should be responsible for choosing an appropriate kind of synchronization—perhaps wait free synchronization, or spin locks, or queuing locks—such that the timing properties that we have *explicitly stated* can be maintained.

What Abstractions are Needed?

A complete answer to the question "What Abstractions are Needed" requires an extensive research program. But I mention here a few examples of places where abstractions seem to be "missing", based on my own experience.

DATA HAND-OFF BETWEEN TWO MODULES

What I am attempting to capture with this heading is an abstraction over procedure call and synchronous rendezvous. What these mechanisms have in common, and thus what the abstraction should capture, is the hand-off of data (the arguments) from an instigating module (procedure or process) to a receiving module, the initiation of execution in the receiver, and the return of (possibly null) results from the receiver to the instigator.

What I would like to abstract away from is the notion of control flow that is tied up with the terms procedure call and rendezvous. When we say "procedure call", we automatically think that it is necessarily the calling process that executes in the called procedure. When we speak of a rendezvous, we cannot avoid implying that the two parties have separate threads.

I believe that we need an abstraction that is agnostic with respect to the implementation decision of how many threads and how much buffering are used to implement the data hand-off. Separating these issues is such a foreign concept that we do not even have a name for the abstraction that I have in mind. Perhaps the term *message send* as used in object-oriented programming comes closest: the common implementation is an indirect subroutine call, but a distributed implementation might well use an inter-process message.

Why do I believe that such an abstraction will be useful? Because it will enable us to describe functionality—the hand-off of data between modules—without constraining the implementation. It should be the business of the compiler, not of the programmer, to decide when it is necessary to create a separate thread and when a subordinate activity can be carried out in the main thread. Of course, this means that the compiler must be ware of the real-time constraints that the program is striving to meet. But that is my whole point: we need ways of being explicit about the behaviour that is important to us, such as response time, while letting the implementation take care of much of the detail.

FIRST-CLASS MESSAGES

The duality of process and message in operating systems is well known (Lauer and Needham 1979). Briefly, the idea is that a system can be structured in one of two ways. It can be process-oriented, where each resource is encapsulated in a monitor and each activity initiated by a client creates a process that travels around the system waiting for and interacting with these monitors. Alternatively, the system can be message-oriented, where each resource is encapsulated in a process and each activity initiated by a client creates a *message* or *event* that is sent to one of these processes. Messages may need to wait to be received by a process; their handling will likely entail the sending of further messages to other processes.

The major influences on the performance of such a system are the cost of the algorithms implementing the functions of the system, the overhead of the primitives, and the time that a process or a message spends waiting in a queue. The costs of the algorithms are the same in both models, but the overhead of the primitives is influenced by the hardware and its architecture; this is the primary reason that one of these

structures may be preferred to the other. With respect to scheduling, Lauer and Needham write:

> *...the two models behave identically with respect to the scheduling and dispatching of client processes. That is, if the message system implements a particular discipline for the queuing and unqueuing of messages, then the procedure-oriented system can implement exactly the same discipline for its queuing and unqueuing of processes. Similarly, if one system forces a context switch in a particular circumstance, either as a result of a kernel operation or a pre-emption due to an external event, then the other model can do exactly the same in response to the dual circumstance.*

However, the duality implies that while processes should be the schedulable entities in a process-oriented system, it is *messages* that should be scheduled in a message-oriented system. For example, if we apply this lesson to a real-time information flow system, we should be seeking to make information flows the schedulable entities, rather than the processes that manipulate those flows.

We see our theme re-appearing: we need the abstractions that let us express directly the features that are important to the application, rather than having those features emerge, as if by accident, as a consequence of a collection of other implementation decisions. It is well known[1] that the single factor that has the most influence on the real-time performance of a real-rate application (such as one streaming audio or video) is the amount of time that data packets spend sitting in buffers. But the typical program contains no explicit mention of that time, and the language and environment provide no knobs to turn that will adjust it. Instead, the time that a packet spends waiting in a buffer emerges as a consequence of collection of other decisions, such as the process scheduling discipline, and the number, size and location of buffers.

SCHEDULING

In addition to abstracting scheduling away from the scheduling of processes to the scheduling of *things*, we also need better abstractions for scheduling. Most commercial systems still use priority scheduling, in spite of widespread recognition that priority is not the right abstraction. In addition to any other deficiencies, priority is not compositional, and it also suffers from priority inversion.

Deadline scheduling has been used quite successfully in many research systems, but it is not clear that appropriate deadlines can always be found in all application domains.

FAULT TOLERANCE

As systems that interact with the real world, fault tolerance is a major concern of embedded systems. What abstractions do we have for dealing with faults and failures?

We should note at once that linguistic mechanisms like exception handling are not designed for handling failures: they are appropriate for transmitting “exceptional” return values when the state of the system is precisely understood, and still within the specification of the component that is raising the exception. To put it another way: if a failure can be anticipated and explicitly coded for, then doing so transforms the failure into an exceptional (non-failure) event. True failures are the cases that cannot be anticipated, and which the system must therefore tolerate without exact knowledge of what has gone wrong.

---

1. Well know to those experienced in this area; I thank Jonathan Walpole for this observation.

Outside the domain of embedded systems, the transaction concept has been very successfully used both to mask and to recover from failures. But transactions are clearly not the right abstraction for a real-time embedded system, because they introduce unbounded delay and the possibility of abort.

This leaves open the question of what an appropriate failure abstraction might be. Note also that the introduction of adaptive and dynamic resource scheduling inevitably gives rise to a new class of failures: overload conditions, when the available resources are simply insufficient to meet the needs of the applications. Handling overloads gracefully is a prerequisite to the wide-spread adoption of more dynamic forms of resource allocation. Thus, finding suitable abstractions for fault tolerance is doubly important.

Summary

It is clear that 30 years of research in operating systems and programming languages have contributed little to solving the problems of embedded systems. The abstractions that we have provided in those disciplines are often inapplicable to the needs of embedded systems, either because the common implementations are profligate in their use of resources, or, more fundamentally, because the abstractions abstract away from the properties most important in an embedded system. First among such properties is the timing behaviour of the system.

Hence I am calling for the creation of an appropriate set of abstractions, abstractions that will let designers of embedded systems specify the properties with which they are most concerned directly, rather than having those properties emerge implicitly.

References

Lampson, B. W. (1983). “Hints for Computer System Design.” *Proc. 9th ACM Symp. on Operating Systems Principles,* Bretton Woods, New Hampshire, ACM Press. pp 33–48.

Lauer, H. C. and R. M. Needham (1979). “On the Duality of Operating Systems Structure.” *Operating Systems Review* **13**(2), pp 3–19.