Languages and Tools for Real-time Systems: Problems, Solutions and Opportunities *

Richard Gerber
Department of Computer Science
University of Maryland
College Park, MD 20742
(301) 405-2710
`rich@cs.umd.edu`

University of Maryland Technical Report
CS-TR-3362, UMIACS-TR-94-117

October 14, 1994

Abstract

This report summarizes two talks I gave at the *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, which took place on June 21, 1994, in in Orlando, Florida. The workshop was held in concert with *ACM SIGPLAN Conference on Programming Languages Design and Implementation.*

The first talk ("Statements about Real-Time: Truth or Bull?") was given in the early morning. At the behest of the workshop's organizers, its primary function was to seed the ongoing discourse and provoke some debate. Besides asking controversial questions, and positing opinions, the talk also identified some several fertile research areas that might interest PLDI attendees.

The second talk ("Languages and Transformations: Some Solutions") was more technical, and it reviewed our research on program optimizations for real-time domains. However, I tried as much as possible to revisit the research problems raised in the morning talk, and present some possible approaches to them.

The following paragraphs contain the text from my viewgraphs, laced with some commentary.

Since so much work has been done in real-time systems – and even more in programming languages – my references are by necessity incomplete.

*Author supported by ONR project N00014-94-10228, NSF grant CCR-9209333, and an NSF Young Investigator Award CCR-9357850.

1 Introduction

One often finds a gaping chasm between those who write research papers about computer systems, and those who earn their paychecks building them. When representatives from both sides attempt to discuss their work, they find each other speaking in strange and foreign tongues. This can be a very frustrating experience for both parties involved.

The lower one steps down the ladder of abstraction in development technologies – say, from ML to C++, to C, to machine-code debugging, to using logic analyzers – the wider the chasm gets. And since real-time programming is still, sadly, *practiced* at a fairly low level, the gap of discourse between University researchers and industrial programmers can at times be wide indeed.

After all, *our* job is to help create higher levels of abstraction – and in so doing, we occasionally have to make assumptions, simplifications, etc. If we had to consider each minuscule step of the problem we would never get anything done. But sometimes practitioners believe that we oversimplify the problem at hand, and sometimes – admittedly – we do.

Their job, on the other hand, is to get the system out before the release deadline slips too far. This may mean replacing a nicely structured C++ class hierarchy with a tight, hand-written, assembly language module. Or forgetting about fancy (but non realtime) debuggers, and turning to an in-circuit-emulator (or ICE). In other words, if it works, and it helps expedite the process, so be it.

Having been one of *them* in my previous career I can assure you that this type of ad-hoc, flexible expediency yields better marginal results than most strict development processes, formal approaches, expensive OO structuring tools, CASE tool suites, high-performance compilers, etc. What I mean is the following. Given limited development resources, and the choice between fancy software processes on one hand, or an ICE, a logic analyzer and a very good consultant/hacker on the other, I'd always take the latter.

However my option is replete with gaping limitations. First, it depends on excellent programmers who can keep track of ad-hoc changes. Also my expert "consultant/hacker" (if one can be found) will not stay forever – hopefully. Moreover, my option requires buying some very expensive hardware, which may not even be upgradable to next-year's chipsets.

But more importantly, the ad-hoc option simply does not scale to modern systems. In multiprocessor platforms, for example, a single-pod ICE will be of little help, and it probably won't provide real-time performance – no matter what the vendors claim. Further, RISC processors with caches make this type of debugging very difficult indeed – the logic analyzer provides no register-level information, and in "real-time mode" the ICE provides little more.

Most importantly, my option would be useless in building a very large system, with hundreds (or thousands) of interacting components. And maintaining it would be a nightmare indeed.

But if I can't use a seat-of-the pants approach, what are the alternatives? Should I turn to an object-oriented language – and its associated class libraries – or will the binding problems get in my way? How about off-the-shelf kernels – can I trust them, or should I "grow my own?" And should I buy expensive, graphical CASE tools, or get into object-oriented-design?

And getting down to fundamentals, how can I get my hands on a good timing analyzer, or even an unobtrusive profiler that I can trust? Should I turn on my compiler's fancy optimizer, or will it give me worse – and even incorrect – performance?

I have heard all of these questions from "builders," though they are not always posed as questions. Sometimes they are rendered as opinions ("No, No, "I don't know," "No"), which have grown out of bitter, frustrating experiences.

So, I would like to offer a challenge for this community: We are in the business of designing abstractions and the tools to support them – including languages, timing tools, standardized kernels, compilers, etc. So let's provide solid abstractions to solve some of these problems in a rational manner – without neglecting the all-too-important details. Maybe then "builders" will be able to use what we "researchers" produce, and perhaps we'll end up understanding each other.

In fact, some nice solutions already exist, and I'll attempt to identify several of them. We should be aware of them, and use them if we can.

2 Statements About Real-Time: Truth or Bull?

This workshop is devoted to a relatively young research area: programming languages and tool support for real-time development. And when one approaches a new research area, it helps to investigate its underpinnings, and to appreciate its contemporary practices. It is especially enlightening to question its mythology, and understand how it arose.

Let's investigate a bit. I'd like to toss out a few statements, each of which captures one "school of thought" about real-time systems development. I'll comment on each one, and explain whether I think it's either "true" or "bull." Of course these are my own opinions, and mine alone – if you disagree, so much the better!

"Truth or Bull?"

Statement 1: *Neither Ada nor POSIX 1003.1-4 can be used for RT.* (**T/B**)

Opinion: "C'mon, let this old dog rest already."

In other words it's bull. But when we investigate the statements – both of which I've heard many times – we find two or three dogs that should finally rest.

First, for many years it was fashionable to beat up on Ada – even among those who weren't familiar with it. Much of this stemmed from the heavy-handed way that DOD imposed it on developers. But the fact remains that Ada *is* used in many mission-critical systems, thus belying the myth that Ada83 and real-time are not compatible.

Often what lies behind this myth is a vague confusion between Ada "the language" and Ada "the runtime environment." And if one only peruses a sample set of standards documents, or perhaps listens to the "lure of Ada," one may easily come to believe that Ada is both "language" *and* "runtime." After all, the standards don't really distinguish between the two, right?

Frankly, many features of the "runtime" portion are indeed incompatible with real-time practice. Examples abound: synchronous rendezvous as the primary IPC, minimum (but alas, no maximum) delay construct, lack of good priority-based scheduling, heavyweight memory management, etc.

But one doesn't appreciate the entire story by reading documents, USENET newsgroups or operating systems textbooks. For a long time DOD program managers have understood these problems, and have allowed embedded-systems contractors to use Ada "the language," supported by customized runtime systems. And since the few main vendors of Ada compilers (e.g., Rational, Verdix, Intermetrics, etc.) sell such environments, a *de facto* standard of sorts has emerged. As for Ada "the language," it's just that – a programming language equipped with tasking constructs, package encapsulation and assorted other features. One may argue whether it's good, bad, ugly, cumbersome, etc., but the fact remains that it is used in many realtime systems.

In a large sense, the Ada 9X real-time annex [30] simply codified many components of the *de facto* standard. It supports shared, protected objects (i.e., typed, shared memory segments), asynchronous transfer of control, first-class interrupt handlers and exceptions. As for real-time primitives, it includes timeouts (and not just delays), direct, priority-based scheduling (and ability to change priorities), priority-ceiling support for protected objects, etc.

The "annex approach" to 9X also codified the *way* that embedded-systems developers *currently use* Ada: The base language is more minimal – and far less complicated – while the more heavy-weight features can be *selectively* incorporated. Nonetheless, whether Ada will find its way into non-military domains remains an open question. (A positive sign is the Gnu-Ada project, which will provide universities, small production houses and the like with cheap support.)

All of which brings us to the Posix standards, or more precisely, the POSIX 1003.4 Standard Realtime Extensions. One frequently hears that "Ada can't support real-time." But even more frequently one hears that "POSIX-compliant systems can't support real-time." Who says this, and why?

First, the POSIX real-time standard is an attempt to bring the POSIX "open systems framework" to the world of real-time development. The idea is that if a certified set of realtime features are adopted in Unix systems, then perhaps I'll be able to run your real-time applications on my platform, and they'll perform as you intended them to perform. This should seem like a decent idea to most people. What's so controversial about it?

The problem is that standards policies always step on the toes of "vested interests." Assume that you just spent the last five years of your career building a real-time kernel. Assume that much of that time was spent selling your concept to your research community, customer base, etc. Now along comes a set of standards that mandates a radically different kernel concept. And, no matter how hard you may try, your kernel will never be compliant. Thus if the standard is adopted, your kernel will be obsolete. Think about it: how would *you* feel?

To get to the bottom of this, let's check out the POSIX 1003.4 extensions: multiple, selectable priority levels, fully preemptive processes, fine-resolution timers and alarms, priority-driven servers, and the like. In other words, the POSIX standard seems to shout from its margins the words "Priority-Based Scheduling." Right?

Wrong. And now we finally come to what folks used to argue about (and some still do) – they argued over scheduling policies. You see, there have traditionally been two schools in this: those who advocated preemptive, priority-based scheduling, and those who advocated nonpreemptive, time-based scheduling. The "priority group" employs methods like "rate-monotonic scheduling" [24, 21] (which statically assigns the highest priority to the highest-rate process), the "priority-ceiling-protocol" [29] (which handles priority inversion due to blocked resources), "EDF-scheduling" [24] (which dynamically assigns the highest priority to the process with the earliest deadline), "general static-priority-scheduling" [31] (which accounts for offsets, blocking, and deadlines), and a variety of other policies. This group argues for an easier design process, cheaper analysis tests, more adaptability, etc.

The "time-line group" uses terms like "calendar-based dispatching," "cyclic executive dispatching," [35, 7, 2, 38, 36], etc. Basically, these terms add up to the same approach – a major frame (or LCM, or hyper-period) is created, and all task instances falling within the frame are sorted to execute non-preemptively. This group argues for more dependability, predictability, less nondeterminism, etc.

This isn't merely a philosophical argument, since representatives from both groups have built kernels based on their philosophies. Thus, while the argument is at times about scheduling, it's partially about operating systems. Since the POSIX standard seems to argue for priority-driven scheduling, the second group – on the face of it – loses out.

But this isn't true. Recall that the POSIX standard includes programmable timers, so one can easily build a cyclic executive scheduler on top of a POSIX-compliant system. Just run an executive at the highest priority, and use the timers to trigger the "launching-time" for each task instance. Thus static scheduling and POSIX are hardly incompatible.

Let's look at things a different way. There is a wide scale of complexity in the world of realtime systems. The scale begins with purely periodic tasks, which don't communicate with each other or the environment. Proceeding up the scale, tasks may protect critical sections with semaphores, or perhaps use shared IO devices. Proceeding further, there may be nested, relative timing constraints, inserted delays and like. Then we get to multiprocessing systems, inter-networked systems, etc.

The further one proceeds up this scale, the less "pure" static-priority scheduling applies. So let's assuming that you choose to use static-priority scheduling whenever possible. If your system is a marginally complex one, I bet that it possesses some element of time-based scheduling.

On the other hand, assume that I have a very critical application in which leaving out a "P()" operation could cause a nuclear reactor to melt-down. For the sake of predictability I've decided to use a cyclic-executive scheduler. However, I still may wish to utilize some available CPU time for less critical, non-realtime processes (like screen updates). If there are several such processes, they will have to be arbitrated in some manner, and probably by using priorities. Thus I'll end up using some priority-based scheduling after all.

What I'm saying is that not only has the "ADA/POSIX debate" been a red herring, but so has the "scheduling debate." It's time to put both of these dogs to rest.

But if we have to declare a winner here, it's the the real-time research community. There is now a panoply of industrial real-time kernels, all of which are well-supported, lightweight, and – to a greater or lesser extent – POSIX-compliant. To name three, QNX [14], VxWorks and Lynx-OS support very fast context-switch latencies, programmable timers, multiple-levels of priorities and full preemptivity. These kernels were built using the principles that university-types developed. For example, without the rich theory of rate-monotonic-analysis (now more appropriately called "static-priority analysis"), there would have been little reason for the major features in POSIX 1003.4, or in Ada 9X.

So let's stop recycling these poor, tired, overwrought arguments, and finally declare victory. We've won.

Statement 2: *In order to build a real-time system, one must perform tight, a priori, static timing analysis.* (**T/B**)

Opinion: "Less and less tight with each new architecture. Is it time to declare defeat?"

In other words, "bull." But I suppose I have to explain myself, since at the workshop my remarks were slightly misunderstood. It was noted that in the absence of timing analysis, off-line scheduling analysis becomes impossible, as does time-based compiler optimization, etc. In other words, "we may as well pack up and go home." True enough.

But re-reading the statement after several months, I'll still maintain my position. As in Statement 1, I can simply give "proof by example." I claim that (1) tight, *a priori*, static timing analysis is not carried out, and (2) real-time systems do indeed get built. Since I hope we can agree on (2), I suppose I'll have to substantiate (1).

Resolved: modern architectures are far too complicated to statically derive tight bounds on execution-times for nontrivial codes. The operant words here are *static*, *tight* and *nontrivial.* The word *static* infers that analysis is performed on the program without actually running it. A *tight* predicted time bound is one that will actually be achieved if the program takes its worst-case path. As for *nontrivial*, I'm assuming that the program contains branching structures, memory accesses, perhaps a few (bounded) loops or function calls.

Let's make it easy and assume no preemptions. Still, how tight will your prediction be? Will it be 15% over actual worst-case (that's impressively tight), 25% (still tight), 50% (not so tight, but not bad), 100% (somewhat loose), 500% (very loose)?

What makes this problem difficult? Remember, a standard hardware platform these days possesses hierarchical caches, shared memory cards, an instruction pipeline (or two), register windows, etc.

Consider the cache: can you really predict the exact, maximum number of misses that your program will suffer? Remember, this involves disambiguating all aliases – an undecidable problem in itself. Thus – since loading a line of aliased data can easily replace another – you may only provide a very conservative upper bound. Also, if data and code aren't segregated, then loading a line of code may replace a data line, and vica versa.

As for shared memory, your program may compete for common-bus cycles with other processors. Or even more typically, DMA-driven devices will steal memory cycles here and there.

Let's turn to register windows. Have you even tried to time a "Jump-and-Link/Save" combination on a SUN 4 chip? I have. Without overflow, you'll find it to be one of the fastest control-transfer operations around – one that's measured in nanoseconds. But on overflow it's one of the slowest, with a kernel trap measured in milliseconds. How is that for variance? (The same problem occurs in "Restore.")

Another issue arises with benchmarks – which ones will *your* analyzer use? The vendor's CPU (i.e., Register-to-Register) timings? How did they obtain them? Can they count memory-cycles, pipeline stalls, bus arbitration delays and the like? And can you account for all of the peculiarities inherent to *your* customized platform?

My point is, all of these factors (and especially the interplay between them) will necessarily lead to conservative estimates. Does this mean we should "pack up and leave" the area of real-time? No way. In fact, *any* timing estimate is better than none at all: if I perform my scheduling analysis assuming an overly conservative estimate, the worst that will happen is that my system will end up wasting cycles.

In fact, there has recently been very encouraging work in this area. For example, the tool described in [28] lets one perform analysis at the source-level; while it necessarily leads to a rough estimate, it's a nice start in the prediction process. In [12] a more accurate timing tool is described; it analyzes micro-instruction streams using an abstract architecture description. Getting more refined, the tool described in [37] analyzes instruction timings at the pipeline level, and [22] pushes (conservative) analysis to both the cache and pipeline levels.

And now, several papers in this workshop present even newer approaches, and we should expect to see further progress along these lines.

Yet I'll still maintain that static analysis will always produce conservative results. And if, by some miracle, someone develops a tool that achieves reasonably tight estimates, we need only wait: the next chip that comes along will defeat it.

So what's my point? It's this: let's stop pretending that we have tight, static bounds on our codes – we don't, we never did (even in Z80 days), we never will. Let's instead take a wider view. At the start the design process, we may not have chosen our hardware platform, or even written any code. Yet some "rough" load analysis should be performed to make an informed choices hardware, software decomposition, etc. For this we need better analytic/simulation approaches, perhaps using hybrid deterministic-stochastic models. Then when the platform is selected, and the code is written, static analysis will play a very dominant role. But let's not rule out profiling and monitoring techniques to get measured execution times on the actual hardware. Who knows – perhaps we were far too conservative (leading to gross under-utilization) or – heaven forbid – too liberal.

Statement 3: *Real-time applications require traceability down to the machine-code level.* **(T/B)**
Opinion: "Unfortunately it is still desirable in some systems."

By traceability, I mean that there are (formal or informal) mappings between the follow process stages:

$\boxed{\text{(1) Functional Spec}} \Longleftrightarrow \boxed{\text{(2) Detailed Design}} \Longleftrightarrow \boxed{\text{(3) Source Code}} \Longleftrightarrow \boxed{\text{(4) Machine Code}}$

Here I'm interested in the mapping between levels (3) and (4). Recall the ad-hoc development method I described in my introduction – which traded fancy software processes for expert programmers, an ICE, and a state-of-the art logic analyzer. If your objective is to build a small-to-moderately sized embedded system, and you have a short deadline, I doubt that you can find a better development process.

But in such ad-hoc environments programmers like to "have their cake and eat it too." That is, they usually like writing their programs in languages such as 'C', but – if worse comes to worse – they'll end up debugging at the assembly level. Why is this so? After all, if I'm going to spend all of that money on an ICE, aren't I going to buy one with a symbolic debugger?

Sure I am. But assume I'm attempting to catch a subtle, intermittent race condition. Perhaps (who knows?) the DMA is running wild, overwriting a variable or two – perhaps even overwriting some interrupt vector locations. Debugging at this level often requires tracing though the actual instructions executed by the processor, the memory addresses accessed – and having all this data organized along a time-line. And: I also have to relate this information back to my source code.

Some of my undergraduates could tell you all about it. I teach a kernel-level programming course, which goes by the nondescript title of Operating Systems (CMSC 412). In addition to the standard textbook material, it requires designing and implementing a small, time-sharing kernel for a "bare" 80486 machine. The students start with simple interrupt-level programming for standard IO; they proceed to implement preemptive, time-shared dispatching; they then construct message-passing IPC; next they build a dynamic linker/loader for multi-threaded, multiple applications (including address patch-up); then a mini file-system, and finally a socket-style "windows" package (to get the feel of X11).

This is not an emulated system; moreover, the symbolic debugger is worthless after the project incorporates multiple '.exe' images. Thus the code has to be debugged the "old fashioned way," via intensive detective work. Remember, when a system like this "hangs," it simply goes dead to the world – and the only recourse is to reboot it. Thus the students have to find clever ways to collect enough data before the kernel crashes (which it inevitably does). What does this involve? Usually a mixture of strategically placed "printf" statements, good experiment construction, etc. But some students – the really desperate ones – build their own runtime monitors. These are primitive affairs: they spew out traces of addresses corresponding to selected, executed instructions. However, when the system hangs, they can compare the addresses to those on their ".asm" listings, and find the offending instructions.

Thus these students end up requiring traceability, though they probably don't know its formal definition. But they do know one thing. Without the C to asm correspondence, their tools would be worthless, they wouldn't catch the bugs, and they'd never finish the course.

Statement 4: *Does this mean CISC is preferable to RISC?* (**T/B**)

Opinion: "I'll hedge on this one."

While I believe there is some truth in the statement, arguing the point seems rather futile. After all, this war was fought in the architecture community over 10 years ago – and I think we can agree that RISC won convincingly. Yes, Intel will continue producing faster, denser, warmer and sweatier CPUs based on the x86-based technology – at least for a while, at least until other chips outperform them in emulation mode. And if the RISC experience has taught us one thing, it's this: that limited chip real-estate can be more efficiently spent on caches, pipelines, register windows and the like. These features can dramatically increase average-case performance, which is exactly what we want in our desktop workstations. This principle is cogently explained in Hennessy and Patterson's text [13], so there's little need for me to belabor it here.

But the perspective changes when I'm looking for a real-time platform, especially one that'll host a mission-critical system. Yes, I do want my general-purpose workstation to perform like a Lamborghini – and maybe I'll settle for a Corvette – to borrow the analogy made in [13]. But my idea of a good real-time CPU is more like a Mercedes Wagon – high class, yet safe and reliable.

Indeed, the very features that enhance RISC performance (loads of registers, pipelines, intense compiler optimizations) can become liabilities in the world of real-time.

As I've mentioned, many embedded systems developers still have to carry out machine-level debugging; thus they demand a degree of traceability from source programs down to the compiled machine code. And as my students have discovered, this process is sufficiently painful, even on Intel-based CISC systems. However, this level of debugging can become pure torture on many RISC machines.

Why? For one, reading RISC code – and tracing it back to the source – can be a puzzling exercise. After all, the compiler is partly responsible for enhancing the performance of a RISC processor. This often implies registerizing as many variables as possible, and then relocating the instructions to keep the pipeline moving. The net result is that the textual relationship between source code and assembler is no longer maintained – indeed, a good compiler tends to place the instructions where you're least likely to find them. Added to that is the complication of deferred execution. Unlike in your typical CISC processor, now the human debugger has to be cognizant of the instruction pipeline's behavior. So, even when critical assembler instructions are located, their textual order doesn't necessarily correspond to their execution order. This factor is especially important when unravelling a tricky race condition where – for example – a DMA and the CPU are intermittently overwriting shared variables. In fact, performing low-level debugging on a complex RISC CPU can be a uniquely unpleasant experience.

Nonetheless, to argue against RISC is to argue against reality. The only way to address this problem, once and for all, is to minimize the need for machine-level debugging. To be sure, most general programmers are able use powerful debugging tools (such as dbx). So, I'd like to offer a proposal to this community. We need similar types of symbolic debuggers, constructed to solve the problems we face: race conditions in multi-threaded environments, fine-grained timing monitors (postulated by "assert" conditions), a way to account for intrusive devices such DMA – and all of it to work in real-time. Are we up to it?

Statement 5: *Compiler transformations/optimizations shouldn't be performed on real-time code.* **(T/B)**

Opinion: "It depends ..."

Many of the points I made in discussing Statements 3-4 seem to argue against compiler transformations of any kind – especially those that involve instruction reordering. To make this somewhat more vivid, consider the simple example in Figure 1. Since the two instructions don't induce dependences on each other, it appears like the safest compiler transformation that one could desire. Right?

Not necessarily. Actually, the variable names hint at the problem. A real-time application

/* Before Transformation */	$\Longrightarrow$	/* After Transformation */
x := x_in_a		x_out_a := f(y)
x_out_a := f(y)		x := x_in_a

Figure 1: A Simple Transformation – Is it Safe?

typically communicates with various external devices, e.g., sensors, actuators, displays, etc. [1] And at least in lower-level programs, you'll still find memory-mapped IO to be a very popular means of managing the interface between the software and the devices.

Thus a reordering transformation may easily introduce a nightmarish bug into the program. Most trivially, consider the situation where "a" is a one-place buffer, in which the reference to "x_in_a" corresponds to reading from the buffer, and where writing to "x_out_a" allows the bus to refill the buffer. I don't think it's necessary to describe why the transformed program is incorrect!

This is an obvious case of a "bad" program transformation, and it's a fairly trivial one at that. But there are many cases that are not at all so obvious; indeed, they can be quite subtle. Revisiting the discussion concerning RISC CPUs, compilers often obtain enhanced performance by "registerizing" as many variables as possible. But if a variable corresponding to an external input port gets stored in a register, the result is – well, probably not what the programmer intended. Sure, IO-mapped variables can be declared "volatile," but even this won't be sufficient in the example above.

We're now touching on the real reason why many real-time programmers are resistant to compiler optimizations in general. First – as we've already stated – their tradition is one of low-level programming, of intimate familiarity with the code and its interaction with the hardware. Abstractions that remove them from the hardware level induce a fear that their programs won't perform as intended. And often this conservatism is well-founded, since many of these programmers have found that debugging their code also involves debugging their compiler – or at least understanding its code generator significantly more than they desired. This is even true when *no* transformations are applied, so you can well imagine their touchiness about aggressive optimizations. After all, they ask, whose code gets produced – theirs or the compiler-writer's? And all too often compiler writers forget (or conveniently overlook) an essential fact: that source codes are written by *human* programmers. So please note: programmers of production-quality, real-time systems will simply not accept a compiler technology that "outsmarts" them, and possibly "disobeys" their intentions. They believe that they understand their code more intimately than a compiler ever will, and it's obvious that they do.

[1] If this were not the case the application wouldn't have any real-time constraints!

/* Before Transformation */	⟹	/* After Transformation */
input(IN_A, &x)		z := g(t)
if (p(x))		**input**(IN_A, &x)
z := f(x)		**if** (p(x))
else		z := f(x)
z := g(t)		**output**(OUT_B, z)
output(OUT_B, z)		

Figure 2: A Speculative Transformation – Good or Bad?

I am not proposing a return to the good old days of assembly-language coding – no way. Even with the bugs and vagaries that have to be conquered and demystified, I'd much rather use a compiled, high-level language to support my software. It would be beneficial, however, if my compiler maintained the intended semantics of my original program.

On the positive side, transformation engines, if properly used, can be of enormous benefit in helping to tune my system. Under the old fashioned technique of assembly-language coding, programmers were much more closely aware of resources consumed by their code. Conversely, the convenient abstractions afforded by higher level languages come at a price. An operation's resource requirements are often hidden from the programmer, and thus may be grossly over- or under-estimated. For example, a function called "doit()" may require 1μs of CPU time, or perhaps 100ms; since we're disconnected from the actual instructions that get executed, we may not be aware of which it is. This level of abstraction can easily lead to a conflict between the timing constraints and the actual execution time. And transformation engines can help to resolve such faults.

Consider the example in Figure 2, where the instruction "z := g(t)" is moved to the top of the program. Assume that there is a tight timing constraint between the two IO operations. Also, perhaps "f(x)" is relatively lightweight and "g(t)" is a real CPU hog. Then it may be the case that the original program cannot meet its constraint, while the transformed program can.

Since it breaks a control-dependence, this type of "speculative transformation" is rarely carried out by general-purpose compilers. Indeed, when an instruction appears within the body of a conditional (but is free of a dependence on it), one should still assume that the programmer had a good *reason* for putting it there. Often the reason stems from a personal coding style, or for the sake of readability. On the other hand, perhaps the reason is more critical; e.g., to avoid an exception. In our example program, there may be an invariant relationship between "x" and "g" that only the programmer understands.

While this seems to again argue against compiler transformations, it is not necessarily the case. Recall the objective to moving "g(t)" out of the program – to help a programmer tune unschedulable code. And production real-time programmers will find this type of code reordering sadly familiar, since it is usually carried out by hand, and often under the pressure of an approaching release deadline. If an *interactive* tool can help automate this process, so much the better. The tool, for example, can help in identifying the "good target" instructions to move, by transferring them to their "correct" places, and by analyzing the results.

3 Program Transformations: Problems, Solutions and Introspections

For the past several years we have investigated the problem of optimizations for real-time programs. (In fact the first report on this work was given at last year's PLDI [15].) I'll now briefly describe the work, how it evolved, and most importantly – what we learned in the process. For those interested in more technical aspects of our work, please see our research papers (accessible by World-Wide-Web and FTP[2]).

3.1 Balancing Time.

Recall Figure 2, in which the original program overloaded its timing constraint, while the transformed program did not. One may be tempted to ask: "Why wasn't the transformed version written in the first place?" The answer is obvious – it's difficult to see the "big system picture" when writing small pieces of its code. Moreover, this may have been a fragment used in many different systems, each possessing their own timing constraints.

But eventually the code's execution time has to get balanced against the system's timing specifications. Often timing bottlenecks are not recognized until fairly late in the development cycle, after the system has been integrated. To achieve balance at this point usually requires a costly and arduous process of instrumentation and system tuning.

By the time we looked at this problem from a research perspective, two trends had emerged. First, a variety of experimental real-time programming languages had been posited (e.g., [17, 20, 23, 26, 34, 11]). While differing in several details, they had converged on a core set of real-time constructs and functionality. Second, there was a proliferation (indeed an explosion) of compiler optimization techniques, designed to take advantage of emerging computer architectures – e.g., RISC, SPMD, VLIW, superscaler, etc.

In a way, compiler optimizations can be considered "tuning instruments" of sorts, in that their objective is to achieve better performance. Therefore, it seemed natural to use similar technologies in hard real-time domains. But this was easier said than done, as we quickly discovered.

[2] Sites listed at the end of this paper.

3.2 Problems to Surmount.

The first problem we encountered was one of semantics – that is, without an unambiguous semantics for a real-time construct, it's impossible to define the notion of a "safe program transformation." Indeed, while there is emerging consensus on a basic set of temporal constraints, languages use various mechanisms to carry them out. For example, timing constructs can be compiled into set of scheduling directives, or perhaps macro-expanded into system calls to be invoked at runtime. These alternatives result in subtle, though important differences in their interpretation.

The second problem was finding the right metric to use in optimizing the code. In high-performance domains, an optimizing compiler's job is to exploit a program's inherent parallelism, and to pack its computations into as many functional units as possible. Thus it enhances average case performance by achieving better instruction-level throughput. But in real-time we are concerned not with enhancing average-case performance, but instead with ensuring adherence to the constraints. In fact we will be satisfied with *increasing* the program's overall execution time – as long as the timing constraints are met.

The third problem was addressing real-time scheduling support. Since "schedulability" often defines whether the constraints will be met, the particular scheduling strategy will play a leading role in optimization metric used. But scheduler support is provided to arbitrate between the demands of *several* programs, while a compiler usually works on one program at a time. This traditional separation of concerns between the kernel and the compiler has evolved for many good reasons, and it is important to maintain.

The fourth problem is shared by all researchers in optimization strategies – for any architecture or objective. That is, we all rely on good data flow analyzers for our work. If it can't be determined whether instruction B depends on instruction A, then reordering A and B is not such a good idea.

We faced two additional issues, both of which I addressed above: timing analysis and traceability. As for the former, optimization for *any* real-time metric will depend on some degree of static timing prediction – even if it only produces a crude identification of CPU-hogging code blocks. As for traceability, optimized – but unreadable – code will be of little use to most systems builders.

3.3 Semantics and TCEL

We embarked on this research with a straightforward, easily-stated objective: to improve a worst-case execution times via modern optimization technologies. The objective was easily to state, but it contained many hidden nuances – and that of language semantics was the by far trickiest. Why?

I've hinted at the essence of this issue – that of transforming Input-Output operations. Most real-time languages implicitly circumvent this problem by imposing the *strictest possible* semantics. For example, "**every** 10ms **do** B," is usually interpreted to mean the block B – as it appears in the source language – is dispatched every 10 milliseconds. But such an interpretation implies that even the most common optimizations would fail the safety test. Why? Since any code within B may correspond to memory-mapped IO, every instruction should be executed in exactly the place it appears. Consider constant propagation: what the compiler may interpret as reading a constant-valued variable may actually be an input function in disguise.

Also, a more insidious problem is that transforming code may actually *decrease* the application's ability to be scheduled. That is, the relocated instructions from one task may steal the CPU time-slot required by another task.

Clearly, then, this model was ill-suited for our purposes. Also, I've already shown the perils of assuming an overly-course-grained semantics. (Many of these perils were also uncovered in our initial work.) So obviously a compromise was in order.

We called our compromise TCEL, for Time-Constrained Event Language. TCEL's syntactic constructs are not unlike those in other languages. However the semantics is significantly different, in that it is based on the time-constrained relationships between observable events. What is an "observable operation?" Any operation that can be detected outside of a single process, i.e., a message-passing operation, an access to memory-mapped I/O, an instruction that induces side-effects on other tasks, or for that matter, a reference to *any* annotated function call or variable. In fact, in TCEL any instruction can be annotated as "observable."

This distinction between "observable" and "unobservable" instructions is the key to our transformation strategy. Since it imposes a "looser" interpretation on the constraints, a compiler can help rearrange the unobservable code to aid in the tuning process.

This semantics was inspired by a principle commonly applied in formal methods. When reasoning about a real-time concurrent system it is often useful to consider only "events of interest," and to abstract away local-state information. Indeed, almost all formal models ease this process by making some distinction between an "event" and a corresponding "action." For example, in Real-Time Logic [18], events are instantaneous – and require no resources – while actions consume nonzero time. Similar distinctions exist in RTRL [5], Timed IO Automata [25], ACSR [19], and in almost every formal approach to real-time. It therefore seemed natural to extend this common technique to a "full-blown" real-time programming language, in which the "events" correspond to actual IO operations within C code.

3.4 Metrics and Transformations

After working out the TCEL event-based semantics, we turned our attention to developing transformation strategies. We focused our attention on two objectives: (1) tuning single programs in isolation, with the result of unconditionally improving the entire application's schedulability; and (2) transforming programs individually, but using a schedulability analyzer to determine whether the optimized programs either improve or degrade performance.

```
L1:  do
L2:     receive(p,&obj_coords1);
L3:  start after 3.5 ms finish within 4.0 ms
L4:     {
L5:     receive(p,&obj_coords2);
L6:     r1 = F(obj_coords1);
L7:     r2 = G(obj_coords2);
L8:     next_cmd = H(r1,r2);
L9:     send(q,next_cmd);
        }
```

Figure 3: A Real-Time Program: Can it be transformed?

After some initial investigation we decided to avoid a third option: inter-task optimizations, where multiple tasks are optimized simultaneously – and where the types of transformations used are guided by the interactions between the tasks. We found this approach to be untenable, as it introduces complex circularities between the runtime scheduler and the compiler. Once introduced these interdependencies almost impossible to break, resulting in a search problem of untenable complexity.

Local Transformations for Feasibility. One type of transformation can always be applied, and it doesn't require additional information about the rest of the system. That is, whenever a task's internal constraints conflict with its *own* execution time, a timing fault would always be the result – regardless of the scheduling paradigm used. We call such tasks *infeasible* (as opposed to unschedulable). Infeasibility conditions can arise when there are nested constraints; e.g., when some deadlines are tighter than periods, or when there are inserted delay statements. When faced with an infeasible task, using local transformations to achieve local feasibility is certainly better than taking no action at all.

Our approach to this problem is to us a variant of of trace-scheduling [6], with the objective of (1) finding the overloaded paths in the program; and then (2) relocating the unobservable code off of these paths.

Consider the TCEL program fragment in Figure 3, which receives sensor data, delays, receives more data; then it transforms the data into a command and sends out the result. The final **send** must take place within 4.0 ms of receiving the original message.

In the usual interpretation of this program, statements L5-L9 would always execute after the delay; i.e., the **after** construct would be "executed" like a "sleep" command in Unix. TCEL's semantics, on the other hand, only induce timing constraints between the three event-triggering instructions: L2, L5 and L9. As for the unobservable statements L6-L8, their execution is bound only by natural control and data dependences.

This looser semantics yields an immediate benefit: if the execution times of L6-L8 conflict with the 4ms deadline, the unobservable code can be moved to help tune the program to its hardware. Perhaps all (or part) of L6 can be executed while the program delays. It may be possible to specialize parts of L7 and L8 to do the same; perhaps some pre-computations can even be executed before L1. In performing these transformations, the observable events act as "semantic markers," denoting the places where code can be moved.

While the method appears straightforward (and in this case simple), using it on larger programs is a nontrivial compiler problem. Indeed, in [9] we show that even in the case of a basic block, simply determining which instructions to move is NP-Hard. And the situation gets significantly more complicated when the program possesses a branching structure, i.e., when the actual execution paths are determined at runtime.

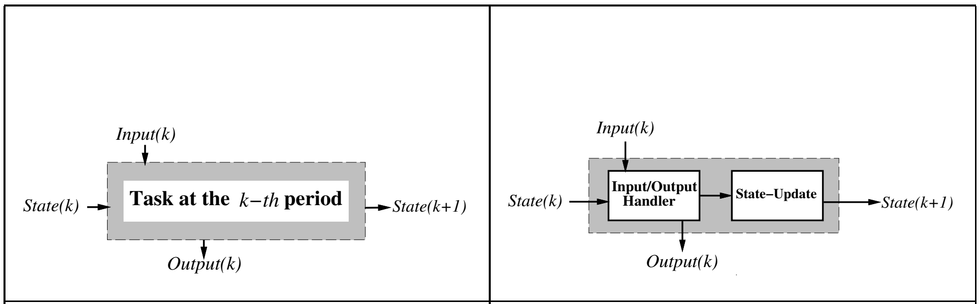
Thus we take a greedy approximation approach, which works in several phases. First the TCEL source is translated into a single-static-assignment (SSA) representation [4], whose naming conventions help isolate the "worst-case" execution paths. Next the code is decomposed into several blocks, and equations are generated to constrain their start and finish-times. Finally a variant of code scheduling is used to relocate the unobservable code, and hopefully attain feasibility.

The actual details of the method are rather complex, and are not within the scope of this paper; readers should refer to our more technical papers referenced below.

Transformations for Schedulability. In [8] we addressed a more ambitious goal – that of transforming multiple tasks to achieve schedulability. The problem here is that "schedulability," by definition, depends on interactions between multiple tasks. Thus, to avoid introducing circularities between the scheduler and compiler, we required the following ingredients: (1) a straightforward means of transforming each program individually, and (2) a simple metric to evaluate whether the transformed version is better than the original.

We narrowed our problem domain a bit, concentrating on control applications with (purely) periodic tasks. There is a wide variety of priority-based schedulers which can support such applications; moreover, most of them also possess cheap, offline analysis tests. This helped satisfy requirement (2) above.

As for requirement (1), common sense dictates that if a task's deadline is increased – without changing its semantics – the entire application's potential for schedulability can be enhanced. In fact this is the motivation behind our transformation strategy. Whenever an application is found unschedulable, we attempt to identify tasks that contribute to the timing problems. Each of these tasks gets split into two threads – one containing its observable events, and the other containing its unobservable instructions. While the former thread must finish by the original deadline, the latter is allowed to "slide" into the next frame. Thus the transformation effectively increases the original



(A)

```
every 25ms
/* Original Task τ */
  {
  receive(Sensor, data);
  if (!null(data));
    {
    t1 = F1(state);
    t2 = F2(state);
    t3 = F3(data);
    t4 = F4(data);
    cmd = t1 * ( t3 + t4 );
    send(Actuator,cmd);
    state = t1 * ( t2 + t3 );
    }
  }
```

(B)

```
every 25ms
/* Slice τ_IO */
  {
  receive(Sensor, data);
  c = !null(data);
  if (c)
    {
    t1 = F1(state);
    t3 = F3(data);
    t4 = F4(data);
    cmd = t1 * ( t3 + t4 );
    send(Actuator, cmd);
    };
/* Slice τ_ST */
  if (c)
    {
    t2 = F2(state);
    state = t1 * ( t2 + t3 );
    }
  }
```

Figure 4: Slicing for Scheduling: Original Structure/Program (A) and Transformed Version (B)

task's deadline, while maintaining its semantics. The pictures in Figure 4 (top) show the effect of the transformation on the behavior of the task's kth period.

The actual transformation is carried out via program slicing. Briefly stated, a *slice* of a program P with respect to a program point p and variable v consists of P's statements and control predicates that affect v at point p. (See [8] for details on our approach; refer to [33, 16, 27, 32] on program-slicing in general.) Consider the program in Figure 4(A), and its two residual slices in Figure 4(B).

Since slice τ_{IO} contains all of τ's IO operations, the observable behavior of the transformed task will be semantically identical to the original.

So the question is: when should a task be sliced, and how can we evaluate the transformed task's effect on schedulability? In our paper we presented a dynamic variant to rate-monotonic analysis, which succeeded in answering this question. This solution was somewhat unsatisfying – at least from our point of view. The offline analysis was indeed cheap to perform, and it allowed the transformation tool to help tune tasks prioritized in rate-monotonic order. However, the on-line scheduler was a bit more complicated than desired, in that it added a dynamic, time-based component to the dispatcher.

But research moves quickly in this field! After investigating TCEL's "delayed deadline" model, Alan Burns at the University of York [3] developed a *static-priority* scheduling scheme to handle such applications. Unlike "pure" rate-monotonic schemes, here the actual priority assignment is dependent the respective execution times of each IO-handler and state-update component, as well as the task periods. Based on his analysis, Burns presented a search algorithm that finds a schedulable static-priority order – or determines when no such order exists. In other words it's optimal.

This analysis, in turn, allowed us to make further improvements. Burns's algorithm assumes that tasks in the application have already been transformed into their two component sub-threads. On occasion, it may be beneficial to slice all tasks first, and then assign their priorities. But we have found that slicing a task frequently adds too much overhead to make it worthwhile; moreover, the resulting state-update component may be quite small. Thus it's a better idea to selectively slice tasks, as necessary. So the question is: which tasks should be sliced?

Burns's analysis provides the answer, since it yields a sound means for evaluating schedulability for arbitrary tasks – whether they are are sliced or not. Based on the analysis, we were able to develop an algorithm that determines the minimal subset of tasks to be sliced – and then slices them. The algorithm is currently driven by an X-interface, and it works in concert with the GNU C compiler and GDB debugger. The tool provides graphical, user-selectable transformations, so that the programmer remains informed of all slicing decisions. In this manner, the programmer maintains a reasonable degree of traceability to the original program. Much of the interface, as well as the slicing routines, was cannibalized from Agrawal and Spafford's Spyder toolset [1]. This is a slick piece of code, and those who are interested in building analysis engines for C modules may wish to check it out.

3.5 Lessons Learned at The School of Hard Knocks

Many of the points made in this paper actually grew out of lessons learned from this research. At the risk of belaboring these points, I think it's appropriate to reiterate them, and how they relate to our work.

Lesson 1: The Limits of Dependence Analysis. Our approach relies heavily on contemporary compiler methods, including intra- and inter-procedural data and control analysis. And as with all program transformation algorithms, the limitations of this enabling technology become a constraining factor of our approach. We originally assumed, rather too eagerly, that dependence-analysis for our types of transformations would be quite simple, especially compared to that required by parallelizing compilers.

This was not the case, and we ultimately suffered the same difficulties as all other researchers in the global-optimization business. While the field of dependence analysis has made rapid progress, sadly the newest techniques are oriented mainly for detecting parallelism in FORTRAN codes. On the other hand, C is the base language of choice in real-time domains. And while FORTRAN and C do share some common characteristics, they differ significantly in many aspects – and most prominently in their styles of indexing arrays. For example, it's difficult to write a C program that that accesses 2-D arrays, and yet avoids using pointers. But as we know, determining pointer-aliased dependencies is one of the hardest compiler problems of all!

One can partially offset this limitation by rewriting (linear) array accesses with FORTRAN-style, integer indexing; by using annotations to give "hints" to the dependence analyzer; or by simply taking very pessimistic dependence sets. None of these techniques is very satisfying. Thus, we're eagerly awaiting more progress on this very challenging problem.

Lesson 2: The Limits of Timing Analysis. Another limiting factor is the difficulty of achieving accurate, static timing analysis – a problem I've already addressed at some length. We found very early in the game that it was futile to predict the execution time of a single instruction (or a relatively small block of code). First, the CPU time is probably too small to make a difference in achieving schedulability, and second, the "noise" in the prediction will be too large.

Thus we have adopted a hierarchical abstraction approach to deal with time predictions. In many cases, we need only account for the CPU-intensive function calls that perform complex operations, and we can ignore the execution time of finer-grained instructions. The same approach is used on larger-grained structures within the control flow graph. Our experience shows the transform engine should usually hunt for the "big-game targets," and forget about the smaller ones.

However after the code is transformed, it becomes imperative to verify the result with a more sophisticated timing tool; for example, a good profiler. Performing such re-timing is especially important in a cached memory structure, where code scheduling will always change the instruction alignment.

At the workshop, it was remarked that this approach (and its resulting algorithms) represents a departure from our original PLDI paper [15]. This is correct. Throughout the course of the project, we gradually learned that that the original approach – while theoretically sound – was untenable to implement. The revised approach (and algorithm) is described in a more recent report [9].

Lesson 3: Traceability is a Must. The results of multiple SSA-based transformations are, at best, cryptic. And as mentioned above, programmers of real-time systems will not accept any auto-transform tool. Since they may eventually resort to low-level debugging, they require – at the very least – iterative feedback during each transformation stage. Moreover, transformations based on the TCEL's event semantics are not fool-proof if the programmer overlooks declaring IO operations as "observable" – which may easily happen with memory-mapped IO. We've already shown (in Figure 1) the ramifications of reordering such instructions.

However, while programmers will reject multiple, automatic transformations, they will enthusiastically embrace a tool that helps tune their systems. These considerations argue for a front-end that permits the programmer to interact with the tool during code optimization process. With a TCEL-style transform engine as its foundation, a graphical interface allows a programmer to selectively apply the transformations – and also remain informed of the results. Hence our current work on the X-interface. Also, the types of transformations we use (i.e., "structure-based") differ from those described in the original PLDI paper. Now their results are suited for humans (and not just computers) to read.

Lesson 4: There's No Substitute for a Good Design. When the design itself is the product of ill-considered timing estimates, transforming the individual tasks will be of little use. Indeed, no amount of system tuning can help in this case – and the only resort is to redesign the system. How can such errors arise? Easily.

Consider the process of designing a real-time system. If it's a control system, for example, functional correctness can be checked at the subsystem level. Moreover, high-level timing requirements are also be evaluated – and sample/update parameters can be adjusted to ensure stability criteria, correct signal-to-gain ratios, etc.

Then the subsystems are decomposed into "tasks," so that the real-time schedulers can tractably guarantee the system's requirements, while also maintaining its functional correctness. This decomposition introduces a new class of intermediate timing constraints, which are artifacts used to realize the original high-level requirements. Perhaps groups of tasks are allocated to processors, based on locality considerations.

And herein lies the obvious problem. It takes a "guru," a very rare kind of specialist, to carry out this process in a manner that balances resource consumption against requirements. It demands someone with much more skill than my high-paid consultant/hacker. In fact given a large enough system it's an impossible task, even for the most seasoned engineer. And so we end up with scenarios like the following: The guru – to the best of his or her ability – mandates that functional units X and Y execute with periods 65ms and 20ms, respectively. But then the programmers code up the system, and find that Y grossly over-utilizes its CPU; further, they discover that most of X's outputs are not being read by the other subsystems. Then the guru readjusts the rates – re-checking them with the requirements. This process may continue many times, until perhaps it converges. During this process the system is tuned, re-tuned, etc. But ultimately, in the worst case it may be determined that a redesign is in order.

And I believe that this is the true problem that we should be addressing – the problem of design refinement. It embraces the areas requirements engineering, design theory, programming languages, timing analysis – all of it. In fact, the PLDI community seems to have a handle on many of these areas. But perhaps it's time to stop treating them individually, and start studying them as a whole. In doing so, we may learn new ways of evaluating design alternatives – before it's too late.

We have done some research on this problem, and our initial results are in a paper entitled "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes" [10]. These results are encouraging, and they do yield a new way to approach the problem – as well as some algorithms to help solve it in certain circumstances. But this paper, and indeed no paper, will *ever* provide *the* "solution to the problem." The problem of real-time design will never be solved as such, only improved.

4 Honesty, Understanding and Opportunity

I initially promised controversial, provocative statements; however, in re-reading my viewgraphs after several months I'm afraid I failed to deliver on my promise. Simply put, much of what I have written thus far should be taken as common sense these days.

But I promised some controversy, and I feel duty-bound to deliver. So let's finally return to the issue of dialogue, both between "researchers" and "builders," as well as between the "real-time" and "programming languages" communities.

4.1 Honesty may, after all, be the best policy

As I wrote in the introduction, academic researchers face a dilemma: On one hand, our tradition is to work with highly abstract models of the domains we analyze. On the other hand, we often end up developing methods that are perfectly sound with respect to our model, but thoroughly impractical to use in the domain itself.

We researchers in real-time face this dilemma more frequently than many of our colleagues, and we're more likely to suffer its pitfalls. The reason is obvious: If we had to factor in every timing peculiarity of a computing system, we would never rise above analyzing timing diagrams. Moreover, we'd still have to worry about functional correctness, just like our colleagues.

But we should never forget one point: an abstract model's representation of timing behavior will differ in subtle, but important, ways from the system's execution profile. Thus we walk a fine line between being inundated with detail on one hand, and being irrelevant on the other.

Within our transformation research (described in the previous section), we took several gross abstractions – mainly in the area of timing analysis. Along the way we decided that fine-grained timing measurements are not obtainable, and we adjusted our method accordingly. Moreover, we developed the work knowing full well that if the results are to realize their potential, we'll require some additional progress in several enabling technologies. New compiler optimizations often share this property, in that they are limited by the state of dependence analysis. But new optimization techniques can help to stimulate additional work on dependence analysis, and so research progresses apace.

Since abstractions and simplifications are essential in even the most "practical" research project, and since we all write papers on our work, we must end up asking ourselves the following question: What's the best way to write about these simplifications, drawbacks, limitations, etc.?

If we're interested in selling our work to "implementors," the answer may just be: "As honestly as possible." Perhaps one reason they find our work simplistic – and at times inaccessible – is precisely because we happen to gloss over those missing details.

No, we don't try to con our readers, or oversell our work by omitting "fatal flaws." On the contrary, the realtime community has, if anything, been guilty of *underselling* its work – especially in proportion to its impact. The problem is this: We are academic researchers, and we write papers for our colleagues. Thus we expect that they have a high degree of technical sophistication, a depth of knowledge equal to our own.

For example, it rarely disturbs me when a new scheduling result fails to account for context-switch latencies in its analysis – that is, unless the latencies would invalidate the analysis altogether. After all, I've read 100, 200, or 500 scheduling papers, and I know how context-switch times are handled in various classes of algorithms. I have learned how to charge latencies to the tasks – and which tasks should be charged. I'm also confident that if I really, truly desired to do so, I could explicitly add context-switching into the analysis equations.

But consider a "practitioner" – someone who builds real-time systems for a living – for whom this paper is perhaps an entree into the "formal" world of real-time scheduling. Regardless of the result's quality, and its potential use, the practitioner may have the following reaction: "This is a pie-in-the sky dream – look, they didn't even care about context-switch times!"

We've all heard this kind of statement, and we've probably dismissed it as abject ignorance. But ignorance does not necessarily imply stupidity, and the "practitioner" probably knows more about building systems than we ever will. And let's face it: we're partly at fault.

The pity of it is this: The author needed only to add a few sentences and citations, clearly denoting that context-switches were not considered. Perhaps it could have been explained that this was to keep the equations free of clutter, and to center the theory on the problem at hand. Moreover, a few references could have been cited – for example, mentioning that "latencies can be handled in the standard way (e.g., as in [ABC])."

But even if context-switching had been overlooked completely, this wouldn't have been a mortal sin. But it could have been clearly pointed out – laced with a few potential ways of dealing with them. At this point the burden would be on the "practitioner" to search a bit further, to check out references, to develop new techniques, to get initiated.

4.2 But please understand what we do for a living

Will practitioners take up their share of the burden, and meet us half-way? I can only hope so. However, at times I'm afraid that it will take more than honesty on our part to bridge the gap – a gap that seems very wide.

In my conversations with the "builders" of "real-life," embedded systems, I've often detected a misunderstanding of what we do for a living, which at times can border on disdain. Our work is perceived – fairly or unfairly – as being far too abstract, too incremental, too "theoretical" to ever translate into useful, deliverable products.

But while we do not build finished products, polished and ready for market, that doesn't mean we are totally irrelevant either. If a scheduling algorithm fails to factor in context-switch overhead, it doesn't imply that it's useless! And even if the technique possesses a mere grain of an idea that can help you build a system, then why dismiss it? Why not give the author a call? If it were me I'd be flattered – and I'd try to help you refine it into something you could use in a "real" application. Let me tell you a little secret: we pie-in-the-sky researchers like nothing better than being useful in this world.

Transforming results into products can be a very long process. But research transitions can be lightening-quick. And regardless of what corporations do, we researchers can certainly learn from each other. Sadly however, the schism between "builders" and "theorists" has even infected us – to the point where we may avoid our colleagues' conference presentations when we consider them irrelevant to our work. We've all seen groups of "practitioners" disappear during the more "theoretical" talks – and vice versa. Do we really have nothing to learn from each other?

I think not. In fact, our research on real-time programming languages grew out of similar concepts found in formal methods. The end-to-end design work had its principal genesis in scheduling theory, not design theory per se. Some of our recent ideas on automatic verification stemmed from presentations on program dependence analysis. The motivation for our parametric scheduling techniques derived mainly from Bill Pugh's work on variable elimination. And so on.

In fact, my Ph.D. work on formal specification grew mainly out of my experiences as a developer, and out of a desire to "find a better approach." But these tales, and those of my fellow "gurus," will have to wait for another paper.

4.3 PLDI + Real-Time = Opportunity

The rationale for this workshop is, I believe, for two separate fields to trade ideas, to learn from each other, and perhaps to start new research collaborations. That said, let me offer my final "true or bull statement."

Statement 6: *There aren't fertile problems for PLDI researchers in realtime systems; moreover, the real-time community is too small, and not worth the bother of getting involved.*

Opinion: "Complete bull on both points."

The second point is easy to address, since the real-time the real-time community is huge, especially when one considers industrial folks who participate. Consider: real-time applications include manufacturing control systems, avionics systems, patient monitoring and imaging equipment, communications and automotive systems. And we can't forget the hottest stuff of all – interactive graphics, animation, mixed media, virtual reality, etc.

I believe that PLDI researchers can get into these areas, and yet still do what they do best – i.e., language design, program analysis, compiler techniques, runtime support, etc. Due to to two converging trends, now is the time for SIGPLAN researchers to leap in.

The first trend has to do with demand. Practitioners now understand that traditional development processes do not scale to modern, complex systems. The low-level approach, when applied to a large system, inevitably ends up in the long, tedious phase of system integration – every manager's worst nightmare. There is now a strong desire to use standardized kernels, to reuse verified modules in many different systems. Most of all, if a project is to survive, it's now essential to avoid that drawn-out process of tuning, instrumentation, etc.

And then there's the second trend. That is, real-time developers are no longer only embedded controls engineers, with fifteen years of highly specialized experience. They are also animators, physicists, video producers, musicians, medical researchers, graphic artists, etc. Programming is probably a very small part of their job. Perhaps they can write C functions, or Hypercard scripts – but these are not real-time systems gurus, nor do they ever wish to be. Yet they struggle with the same issues that"hard-core" developers would find familiar. These issues usually deal with with doing the following activities within a predictable amount of time: (1) getting data into the system from a device; (2) processing it in some manner; and then (3) dumping the results out to another device.

Indeed from this perspective, most contemporary developers are real-time programmers to some extent – yet these developers still have to resort to very primitive design methods. Just ask any animator about the "science" involved in laying out a Quicktime-compressed video on a CD. One will usually get a response such as "I tried compression X, then I tried Y; then Z; I wrote the it out on the CD at 40 frames per second; then 30. Well, I guess it looks OK to me, sort of."

And herein lies an opportunity. We now have solid theoretical foundations on which we use to develop new software technologies. Moreover, hard real-time systems engineers will eagerly accept any good alternatives to the familiar, labor-intensive implementation methods. Most of all, we have a new class of implementors who possess neither the qualifications nor the time to learn the traditional methods. They demand new approaches to help design, implement, and integrate their systems – and the demand is growing every day.

The time is right for this community to help satisfy their demand.

For More Information. The topics I've discussed in this paper draw on material in several fields – real-time systems, programming languages, software engineering, formal methods, design engineering and others. My references are by necessity incomplete, and not at all representative of the work performed in any of these fields. Many more references are given in my group's papers on these subjects, available via the World-Wide Web at:

`http://www.cs.umd.edu/projects/TimeWare/TimeWare.html`

or via FTP at:

`ftp.cs.umd.edu:/pub/realtime`

The ACM SIGPLAN Real-Time Workshop '94 also has its own home page, accessible via WWW or ftp:

`http://www.cs.umd.edu/~pugh/sigplan_realtime_workshop_94`
`ftp.cs.umd.edu:/pub/faculty/pugh/sigplan_realtime_workshop_94`

Finally, the IEEE Technical Committee on Real Time Systems has a WWW home page, maintained by Azer Bestavros:

`http://cs-www.bu.edu/pub/ieee-rts/Home.html`

References

[1] H. Agrawal, Richard DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] T. Baker and A. Shaw. The cyclic executive model and ada. *The Journal of Real-Time Systems*, 1(1):7–25, September 1989.

[3] A. Burns, K.Tindell, and A.J.Wellings. Fixed priority scheduling with deadlines prior to completion. In *6th Euromicro Workshop on Real-Time Systems*, pages ??–??, June 1994.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and systems*, 9:319–345, July 1987.

[5] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, method for validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, January 1985.

[6] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computer*, 30:478–490, July 1981.

[7] G. Fohler and C. Koza. Heuristic Scheduling for Distributed Real-Time Systems. MARS 6/89, Technische Universitat Wien, Vienna, Austria, April 1989.

[8] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, December 1993.

[9] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. Technical Report UMD CS-TR-3323, UMIACS-TR-94-90, Department of Computer Science, University of Maryland, July 1994.

[10] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994. To appear.

[11] R. Gerber and I. Lee. A Hierarchical Approach for Automating the Verification of Real-Time Systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, 1992.

[12] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings IEEE Real-Time Systems Symposium*, pages 68–77. IEEE Computer Society Press, December 1992.

[13] John L. Hennessy and David Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 1989.

[14] Dan Hildebrand. A microkernel posix os for real-time embedded systems. In *Proceedings of Embedded Systems Conference*, April 1993.

[15] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM Press, June 1993. *SIGPLAN Notices, 28(6):166-176.*

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graph. *ACM Transactions on Programming Languages and systems*, 12:26–60, January 1990.

[17] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of OOPSLA-90*, pages 289–298, October 1990.

[18] F. Jahanian and Al Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[19] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Apprach to the Specification and Analysis of Resource-Bound Real-Time Systems. *IEEE Proceedings*, 82(1), January 1994.

[20] I. Lee and V. Gehlot. Language constructs for real-time programming. In *Proceedings IEEE Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society Press, 1985.

[21] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[22] S. Lim, Y. Bae, C. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994. To appear.

[23] K. J. Lin and S. Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1988.

[24] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[25] M. Merritt, F. Modungo, and M. Tuttle. Time-Constrained Automata. In *CONCUR '91*, August 1991.

[26] V. Nirkhe. *Application of Partial Evaluation to Hard Real-Time Programming*. PhD thesis, Department of Computer Science, University of Maryland at College Park, May 1992.

[27] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[28] C. Park and A. C. Shaw. Experimenting with a program timing tool based on source-level timing schema. In *Proceedings IEEE Real-Time Systems Symposium*, pages 72–81. IEEE Computer Society Press, December 1990.

[29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Software Engineering*, 39:1175–1185, September 1990.

[30] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual, Draft ANSI/ISO Standard, ANSI/ISO/IEC CD 8652*. Intermetrics, Inc., 1993.

[31] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[32] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[33] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[34] V. Wolfe, S. Davidson, and I. Lee. RTC: Language support for real-time concurrency. In *Proceedings IEEE Real-Time Systems Symposium*, pages 43–52. IEEE Computer Society Press, December 1991.

[35] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

[36] X. Yuan, M. Saksena, and A. Agrawala. A Decomposition Approach to Real-Time Scheduling. *Real-Time Systems*, 6(1), 1994.

[37] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *The Journal of Real-Time Systems*, 5(4), October 1993.

[38] W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling Tasks with Resource requirements in a Hard Real-Time System. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.