Vertically integrated analysis and transformation for embedded software

John Regehr
School of Computing, University of Utah

Abstract

Program analyses and transformations that are more aggressive and more domain-specific than those traditionally performed by compilers are one possible route to achieving the rapid creation of reliable and efficient embedded software. We are creating a new framework for Vertically Integrated Program Analysis (VIPA) that makes use of information gathered at multiple levels of abstraction such as high-level models, source code, and assembly language. This paper describes our approach and shows how and why it will help create better embedded software.

1 Introduction

Embedded software needs to meet stringent requirements such as high reliability, minimal use of resources, and short development time. The problem is that these requirements are not only individually difficult, but are also in tension. This position paper describes Vertically Integrated Program Analysis (VIPA), a partial solution to this problem. VIPA is a new framework that supports high-level optimizations and analyses that are more aggressive and more domain specific than those traditionally performed by compilers. Its distinguishing features are:

- Integration of tools operating at multiple levels of abstraction: the model level, source code level, and binary level.
- Extraction of as much information as possible from each tool, as opposed to the common practice of using program analyses separately to produce binary results, e.g., "the network acceptor thread has the potential for stack overflow."
- Support for coarse-grain transformations such as merging many logically concurrent activities into a few physical threads, rather than traditional fine-grain compiler optimizations.
- The developer gets to keep using a standard embedded toolchain — we are not proposing to replace or even modify the compiler, linker, etc.

Embedded software needs high-level transformations not because they can be used to create systems that are more efficient than one-off software written by domain experts, but rather because optimizations permit developers who are not necessarily domain experts to create straightforward, untuned code while being confident that the eventual runtime system will be efficient. For example, consider a scenario where a developer wants to call a useful library routine, for example one that is part of a signal processing suite. This developer should not have to worry that (1) a lot of dead code will be linked in along with the called routine, (2) synchronization code in the library will lead to inefficiency even when the routine cannot access any shared data, (3) new and unexpected failure modes will be created due to the possibility of numeric overflow, underflow, or divide-by-zero, or (4) the routine contains latent bad behaviors not likely to be discovered during testing, such as executing for much longer than the expected run time when an iterative algorithm fails to converge quickly. Appropriate static analyses and transformations can eliminate the potential for each of these errors and inefficiencies. Figure 1 provides a few more examples of useful analyses and transformations that we hope to eventually incorporate into VIPA.

In the next section we provide further motivation for VIPA; Section 3 describes our approach in more detail. Section 4 gives several examples of vertically integrated program analysis. In Section 5 we outline several research challenges that we are addressing. Section 6 compares VIPA to related efforts and Section 7 concludes.

2 Motivation

Our work is driven by four main observations.

First, information is both lost and gained at each of the three levels of abstraction — model, source code, and executable system. For example, time constraints and mutually exclusive modes that are apparent at the model level are lost in the source code. Similarly, modern compilers have a great deal of flexibility in the translation they perform, making it difficult or impossible to tightly bound execution time or memory requirements without examining compiler output. This gain and loss of information implies that analysis at all levels is unavoidable. For example, it is generally not

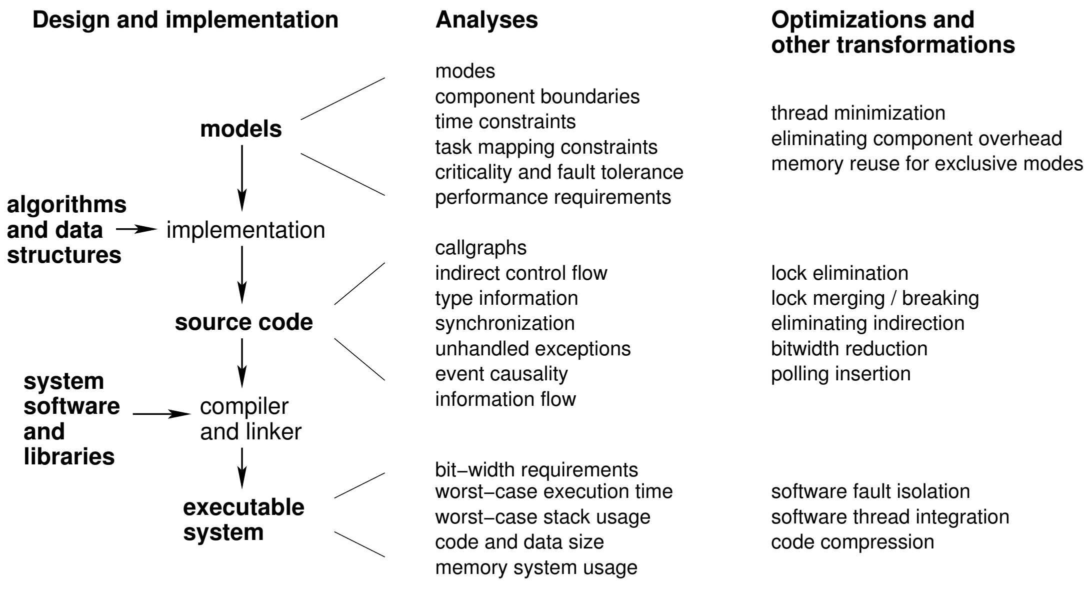


Figure 1. Program analyses and optimizations at multiple levels of abstraction

the case that an extremely sophisticated analysis operating at the model level can supplant an analysis that is properly performed on source code, or vice versa.

Second, software analysis techniques are most often used separately to answer binary questions, e.g. “is the system schedulable?” or “is stack overflow possible?” Although these results are useful, we claim that significant additional benefit can be extracted from analyses by finding ways to interpret them quantitatively. For instance, in earlier work [12] we demonstrated that real-time schedulability can be quantified using a *robustness* metric that indicates the likelihood of deadline misses in the presence of software timing anomalies. Robustness was directly useful in evaluating the desirability of an otherwise useful program transformation: running many logically concurrent activities in a small number of physical threads. Similarly, in another paper [14] we showed that instead of merely using worst-case stack depth information to determine whether a system was vulnerable to stack overflow, this information could also be used to drive a whole-program optimization.

A basic observation is that there is a strong connection between analysis and transformation tools. For example, a tool that detects potential race conditions can be (crudely) turned into a tool for eliminating unnecessary synchronization by attempting to remove, in turn, each lock in a system and testing for races after each removal. Clearly this would not be desirable if the tool were unsound, a topic we return to in Section 5.2.

Third, any program transformation that is performed to improve some aspect of a system usually has side effects on other aspects. For example, representing boolean variables as bits instead of words saves memory but reduces efficiency, and inlining function calls improves efficiency while increasing code size. Furthermore, transformations not only affect metrics of interest, but also potentially enable other transformations. For example, merging two logically concurrent activities into a single thread may expose opportunities to eliminate synchronization operations. Individual embedded projects have particular prioritizations among goals such as minimizing memory requirements, maximizing battery lifetime, etc., motivating our flexible approach to analysis and transformation: it should be easy to make flexible connections between tools, and to change the structure of feedback loops easily, to conform to the changing needs of individual projects.

The final motivation for our work is that program analysis tools are rapidly becoming more useful and practical. This trend is amplified by the large and growing asymmetry between the speed of the typical workstation and the speed of the typical embedded processor. We will enable further exploitation of this asymmetry by developing ways to use multiple analyses and transformations together and in feedback loops.

3 Approach

The VIPA framework begins with standard tools — we do not believe that embedded developers are ready to sacrifice well-understood and well-tested development methods. At the top level, software models are specified using traditional formalisms such as sets of concurrent tasks with periods and deadlines [7], collections of components [16], and hierarchical state machines [5]. Source code implementing the models is written by hand, reused from libraries, and generated by tools such as Simulink. Finally, standard compilers, linkers, runtime libraries, and operating systems are used to create executable systems. In the traditional development method, program analyses may be used to find errors in models, sources, or binaries. Typically these results are interpreted by humans rather than being fed back into the development process.

VIPA augments the standard development process by providing infrastructure for connecting analyses and transformations in feedback loops. This section describes our approach in more detail.

3.1 Exchange formats

Since the idea of a "universal intermediate format" is probably unworkable, we are developing some simple, open formats for exchanging analysis information between tools. Exchange formats will be customized for particular purposes.

One is based on the callgraphs in a system — typically there is a callgraph per entry point. For example, here is the callgraph starting at `main` for a simple embedded system where the colon denotes the "is called by" relation:

```
pulse_task : main
process_char : main
hexdigit : process_char
init_pulse : main
init_timer : main
putc : hexdigit
```

We will augment callgraphs as appropriate to support other information such as lock acquisitions and releases, memory requirements, worst-case execution times, etc. A variant of the callgraph format is our Task Scheduler Logic (TSL) [15], which relates callgraphs, schedulers, and synchronization for a system using a set of axioms about preemption and hierarchical scheduling. TSL can also be used to abstract over locking primitives [13], allowing software components to be agnostic, for example, about whether they are called in interrupt context or in a process context.

Other analysis interchange formats that we believe will be useful will describe:

- The task decomposition of a program from the real-time point of view: at what frequencies are various computations invoked at, what are their deadlines, what are their dependencies, etc.?
- The software component view, relating components and their connectors [16].
- The modal view of the system: which components are mutually exclusive and hence can share reusable resources like memory, and cannot cause race conditions with each other?

We will also develop a library of "helper tools" that will make it easier to track entities (objects, components, function calls, etc.) across multiple levels of abstraction.

3.2 Adapting analyses

In our early experience making tools work together, we found that it was fairly easy to connect existing analyses and transformations. For example, minimizing the stack memory requirements for embedded codes (see Section 4) required work in three steps:

1. Making a stack bounding tool output a list of callgraph edges that are contributors to the worst-case stack depth.
2. Making a global inlining tool accept a list of edges to inline, rather than relying on its built-in heuristics.
3. Writing a script to drive the feedback loop between the two tools, including making calls to the C compiler to generate a binary program to analyze at each iteration of the loop.

This simple feedback loop took a day or two to implement, and would be nearly trivial using the infrastructure that we propose in this paper since each tool would already understand a common data format.

3.3 Changes to the development process

In some cases, such as our experiment in minimizing stack depth, application code does not need to be modified to make use of VIPA. However, the developer does have to create a cost function describing how to trade off between metrics of interest (code size vs. stack depth in our case), and to set up a feedback loop involving several tools. In the long run we envision using a tool similar to VISTA, which was developed by Kulkarni et al. [10] to find effective sequences for compiler optimizations. However, while VISTA is used to parameterize a single compiler, our tool will have to know how to call a multitude of analysis and transformation tools, evaluate the results, and repeat the process until a termination condition is satisfied.

Other analyses and transformation in VIPA will require changes to application source code. For example, to take advantage of optimizations that map many concurrent activities to a few OS-supported threads, software must be written to use a specific "virtual thread" interface, so that they can be linked a non-preemptive application-level scheduler running in each thread. Similarly, to take advantage of concurrency analysis and optimization techniques, use of locks must be visible to analysis tools.

4 Examples

Figure 2 shows two examples of applying vertically integrated analysis and transformation. Figure 2(a) illustrates the example we mentioned earlier where stack depth information drives a whole-program function inlining tool. The feedback loop operates by running the stack analyzer, which identifies the path of function calls leading to the worst-case stack depth — the inliner then tries to inline calls on this path. This transformation often leads to a new program with a different path to the worst-case stack depth, and so the procedure repeats until no further improvements can be found. It was very successful in practice: on a collection of programs for networked sensor nodes, it reduced stack memory requirements to 68% of the requirement when compared to an aggressive inlining policy that is intended to increase performance, and to 40% of the requirement of systems compiled without any function inlining. Furthermore, these benefits were achieved without greatly increasing code size, the usual price of aggressive inlining.

We did not measure the speedup or slowdown produced by the inlining, but we hypothesize that there was modest speedup: unlike code running on general-purpose operating systems, where code bloat due to inlining may cause slowdown due to increased paging or cache misses, code running on microcontrollers should show uniform speedup due to inlining since each inlined function call avoids a substantial number of pushes and pops. The feedback loop in Figure 2(a) could be easily extended to optimize for a third parameter such as code speed. This would require a method for estimating execution times using profile data or the output of a worst-case execution time tool. A slightly more sophisticated variant of this optimization strategy would assign each piece of code a weight equal to its execution frequency, where worst-case frequencies are extracted from the real-time model for a system. This would have the effect of preferentially speeding up frequently-called code, such as interrupt handlers. An even more sophisticated approach would be to integrate a real-time schedulability analysis into the feedback loop. This would cause VIPA to preferentially optimize sections of code that are impediments to schedulability, such as non-preemptive critical sections that contribute large blocking terms to other tasks in the system.

Figure 2(b) illustrates a more ambitious integrated analysis and optimization that we have not yet implemented. The input is a collection of tasks — logically concurrent entities that can be mapped to a smaller number of physical threads. Each task is compiled and subjected to an analysis that determines its worst-case execution time (WCET), an analysis equivalent to finding the longest executable path through the task. Each task is also subjected to a concurrency analysis at the source code level to determine which shared resources it uses and how frequently. The task mapping phase is responsible for mapping at least one task to each preemptible thread instantiated by the embedded operating system. Its criteria for doing so could be

- minimizing the number of threads [18],
- jointly minimizing threads and maximizing the robustness of the system in the presence of timing faults [12],
- minimizing total stack memory consumption, or
- maximizing the number of (static or dynamic) synchronization operations that can be eliminated.

Whatever the policy, it is likely that thread mapping will enable some synchronization operations to be eliminated because all tasks accessing some shared resource will be running in the same thread, and hence sequentially. Synchronization elimination, in turn, changes the structure of the problem by reducing the execution times of some tasks. Therefore, more passes through the thread mapper should permit the task mapping to be refined. Another degree of freedom can be added to the optimization process by supporting lock coarsening [2] — the combination of several short critical sections into one long one. Lock coarsening trades off between responsiveness and efficiency: coalescing several critical sections removes overhead, but also potentially increases synchronization delays experienced by other tasks.

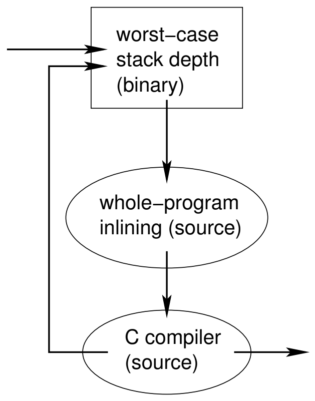
We have presented two examples of vertically integrated analysis: one that we have already shown to be successful and one that we believe will be useful based on our previous experience with thread minimization [12] and concurrency analysis [15], but that we have not yet tried. These examples barely scratch the surface: between the analyses and optimizations listed in Figure 1 and the many others that we did not list, we believe that there are many opportunities for exploring the benefits of vertically integrated analysis.

5 Research challenges

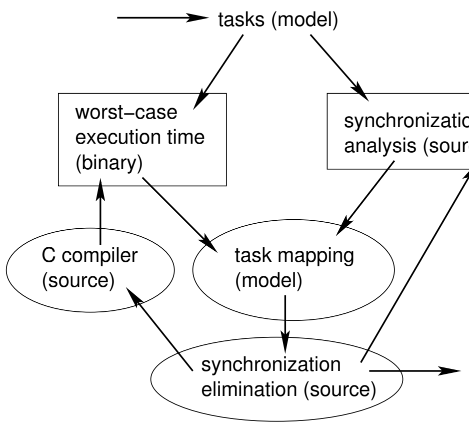
This section describes several research challenges faced by Vertically Integrated Program Analysis.

5.1 Maintaining invariants

Program annotations stored in the exchange formats are fragile in the sense that they may be invalidated during the build process. As a simple example, almost any transformation can change a program's worst-case execution time and worst-case stack depth. However, only a few high-level



(a) Iteratively reducing stack depth



(b) Using synchronization patterns and WCET to drive task mapping

Figure 2. Two examples of vertically integrated analysis and transformation. Analyses are boxed and transformations are circled.

transformations such as lock elimination, lock coarsening, and thread minimization affect the concurrency structure of a system; other tools such as C compilers and tree height reducers [8] should never affect concurrency. The problem boils down to one of characterizing the behavior of "reasonable" code transformation tools. As a first approximation, we will find the set of annotations might be reasonably affected by each transformation, and invalidate them each time that transformation is called. If this turns out to be too conservative then more sophisticated approaches will be necessary, such as spending more effort figuring out if a particular run of a tool invalidates a particular annotation.

A related issue is ensuring that legacy tools can be trusted to behave in certain ways, or to perform certain transformations. For example, our stack-minimization example relies on the C compiler to inline a function call when asked. The GNU C compiler is reasonable in this sense: it can usually be coerced into respecting a set of inlining requests. Similarly, C compilers are generally reasonable in that they preserve function names across the translation. C++ compilers are slightly less reasonable — to track functions between C++ and assembly, VIPA will need to understand name mangling conventions.

5.2 Avoiding bloat in the trusted computing base

Together, the compiler, libraries, operating system, and target hardware form the trusted computing base for an embedded system — a bug in any of these elements can affect the correctness of a product. Embedded developers are rightfully suspicious of new versions of trusted tools such as compilers may be buggy, or may expose latent application bugs through apparently innocuous changes. Asking a developer to add a new tool to the trusted computing base is asking a lot, and for this reason many advanced analysis tools (RacerX, for example [4]) operate opportunistically: they find bugs but do not attempt to guarantee the absence of even simple classes of bugs.

In specific cases tools can be shown to be correct, or their results can be validated externally, but in general developers are probably just going to have to start trusting more software. Unfortunately, it does not seem likely that there is a general technical solution to this problem. In the end the tradeoff is economic: if the net value provided by a tool is positive, developers are likely to eventually adopt it.

5.3 Avoiding long build times

Fast machines and incremental compilation schemes ensure that waiting for compilers is not usually a problem for developers. However, aggressive analysis and transformation techniques threaten to make compilation into a batch, rather than interactive, activity. For example, our early experience with stack minimization showed that connecting an inlining tool, a stack analysis tool, and a C compiler in a feedback loop resulted in a discouraging slowdown — stack minimization took as long as about an hour in some cases. However, our feedback loop was crude: it explored a large solution space, learned only a single fact from each trial compilation, and made no attempt to predict tool behavior.

Clearly we need strategies for speeding build times; we plan to develop techniques for:

- Caching previous analysis and transformation results and invalidating them only when necessary.
- Categorizing analyses and transformations into those that can be performed online, while a developer waits, and those that should only be performed rarely, such as on a nightly test build of a system.
- Predicting the effects of transformations. For example, in the stack minimization example, if we could have accurately predicted the effect of a function inlining decision on code size and stack depth, the run time would have been reduced to at most a few seconds.

Predicting the effects of optimizations is not always easy, and in fact this has been the topic of recent research [21]. We have made progress towards predicting the effect of function inlining on code size using an empirical approach based on least-squares estimation. Early results indicate that while it is difficult to accurately predict the effect of inlining a single call, it is possible to achieve aggregate accuracy across a number of inlining decisions. On the other hand, predicting the effect of inlining on stack depth appears to be more difficult: we saw a number of cases where inlining a particular function call would dramatically increase the amount of stack memory consumed along a particular path. We hypothesize that this happened when the compiler's register allocator became overloaded and stopped performing well.

5.4 Predictability and errors

There are several levels at which predictability is important to embedded developers. First, systems must be predictable in the sense that timing effects can be understood and accounted for. For example, caches and preemption make systems less predictable in this respect. Second, tools must be predictable — developers must be able to understand when and why a tool was able to make a particular transformation, and also why a particular transformation could not be made. For example, consider a performance-critical loop in a hypothetical system based on an advanced type-safe programming language:

> Early prototypes showed that the compiler could eliminate array bounds checks within the loop, permitting a product to be based on a cheap, slow CPU. Then, much later in development it became clear that a slightly more sophisticated algorithm needed to be used in the critical loop. Due to added variable aliasing, the compiler then failed to eliminate the bounds check. At this point, developers had some unattractive options: turn off the bounds check (assuming the compiler permits this), revise the hardware to include a faster processor, or back off to the old algorithm in the critical loop.

This is an example of a *fragile optimization*: one that was not robust with respect to perfectly reasonable changes in the code. Possible strategies for dealing with this problem include:

- Using restricted language constructs that do not have the fragile optimization problem. For example, Kowshik et al. [9] restrict loop indices to always have an affine relation to array sizes, guaranteeing the success of static array-bounds elimination.
- Keeping developers informed as to the true costs of their code, so that they will be aware of potentially fragile optimizations in time- or memory-critical code.

We will also develop ways for chains of tools to provide developers with coherent, high-level descriptions of errors and other analysis results. We believe that development time can be reduced by providing error messages at the right level of abstraction — errors at the wrong level can be very difficult to understand. Providing good error messages can be extremely difficult: the problem stems from the intrinsic difficulty of getting a machine to answer a "why" question: the proper answer may depend on the questioner's state of mind. However, by keeping track of causal relations between actions taken by different analysis and transformation tools, we hope that the VIPA infrastructure will be able to provide some "why" answers.

6 Related work

There is a large body of existing research on analysis and optimization of embedded software, and there are also a number of research projects that perform analyses and transformations at multiple levels of abstraction. For example, the Spring operating system [19] used a sophisticated suite of tools for analyzing and transforming real-time code, and software thread integration [1] combines high-level timing requirements with low-level software representations to compile away fine-grained concurrency. Even so, to the best of our knowledge, our approach is a new one: its goal is to provide an open framework for many different analyses and optimizations, not just a few predetermined ones. This will make it possible for individual project managers and developers to select the analyses and transformations that make sense for particular systems.

A problem similar to the one that we are trying to solve is found inside modern compilers, where there are several program representations at different levels of abstraction. Compiler writers face the difficult problem of performing analyses and optimizations at the correct place, in phasing optimizations in such a way that the maximum benefit is obtained with reasonable compile times, and in moving semantic information between different parts of a compiler. These problems have motivated a number of research projects such as semantics retention [11], VISTA [10], and Gospel/Genesis [20]. There are significant differences between problems faced by VIPA and problems faced inside of compilers. First, while a compiler can specify and modify all code representations, we choose to treat the compiler and some other tools as black boxes. The second significant difference is that while a compiler typically deals with a high level language, an intermediate representation, and machine code, our work addresses the larger semantic gap between models and machine code.

Model-based design of embedded software [6] has goals similar to ours — it aims to look at software from multiple points of view, and to apply multiple analyses. The difference is that model-based design can be viewed as "horizontally integrated analysis," leaving out many useful analyses that occur at low levels of abstraction. As we have shown, low-level analyses such as stack usage and worst-case execution time can usefully feed back into high-level models and analyses.

Finally, a plausible alternative way to achieve the benefits that we hope to provide with VIPA would be to create a sophisticated compiler infrastructure that is capable of meeting many conflicting goals such as those summarized by Dutt et al. [3]. This compiler would almost certainly need to be extensible because Robison's argument that "boutique optimizations" are not economically feasible in production compilers [17] applies strongly to embedded systems, where quirky hardware and highly specialized codes are the rule rather than the exception. In the long run an extensible compiler may be the right solution because it permits aggressive, domain-specific fine-grain transformations in addition to the coarse-grain transformations that VIPA facilitates. However, VIPA has the significant advantage of permitting developers to keep using popular, time-tested embedded compilers such as gcc.

7 Conclusion

By allowing developers to create efficient software without expensive and error-prone manual specialization and optimization, VIPA will make it easier to rapidly develop efficient and reliable embedded software.

Acknowledgments

The author would like to thank Alastair Reid, Ganesh Gopalakrishnan, and Wilson Hsieh for their help in discussing and refining the ideas in this paper.

References

[1] Alexander Dean. Compiling for concurrency: Planning and performing software thread integration. In *Proc. of the 23rd IEEE Real-Time Systems Symp. (RTSS)*, Austin, TX, December 2002.

[2] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, March 1998.

[3] Nikil Dutt, Alexandru Nicolau, Hiroyuki Tomiyama, and Ashok Halambi. New directions in compiler technology for embedded systems. In *Proc. of the Asia and South Pacific Design Automation Conf.*, Yokohama, Japan, January 2001.

[4] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[5] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[6] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proc. of the IEEE*, 91(1):145–164, January 2003.

[7] Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzàlez Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[8] David J. Kolson, Alexandru Nicolau, and Nikil Dutt. Integrating program transformations in the memory-based synthesis of image and video algorithms. In *Proc. of the IEEE Intl. Conf. on Computer-Aided Design (ICCAD)*, San Jose, CA, November 1994.

[9] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proc. of the Intl. Conf. on Compilers Architecture and Synthesis for Embedded Systems (CASES)*, Grenoble, France, October 2002.

[10] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proc. of the 2003 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, San Diego, CA, June 2003.

[11] Steven Novack, Joseph Hummel, and Alexandru Nicolau. A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation. In *Proc. of the 8th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 289–303, Columbus, OH, August 1995.

[12] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symp. (RTSS)*, Austin, TX, December 2002.

[13] John Regehr and Alastair Reid. Lock inference for systems software. In *Proc. of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Boston, MA, March 2003.

[14] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[15] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, December 2003.

[16] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000. Springer Verlag.

[17] Arch D. Robison. Impact of economics on compiler optimization. In *Proc. of the Joint ACM Java Grande/ISCOPE 2001 Conf.*, pages 1–10, Palo Alto, CA, June 2001.

[18] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp. (RTSS)*, Orlando, FL, November 2000.

[19] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The Spring system: Integrated support for complex real-time systems. *Real-Time Systems Journal*, 16(2/3):223–251, May 1999.

[20] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[21] Min Zhao, Bruce Childers, and Mary Lou Soffa. Predicting the impact of optimizations for embedded systems. In *Proc. of the 2003 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–11, San Diego, CA, June 2003.