Compiling for Resource-Constrained Platforms using ASSHes: Application- and System-Specific Heuristics

Alastair Reid Nathan Cooprider John Regehr
School of Computing, University of Utah

Unpublished draft — November 2004

ABSTRACT

We present and evaluate application- and system-specific heuristics (ASSHes) for optimizing embedded software. ASSHes are scripts that decide which compiler optimizations to apply; they make decisions using feedback from previous compilations, using the results of program analyses, and using high-level information, e.g., "there are only 4 KB of RAM, and interrupt handlers 5–7 should be fast since we expect them to execute very frequently." In contrast, traditional compilers base optimization decisions on built-in heuristics that are oblivious to high-level application- and platform-specific concerns. The ASSH effectively becomes the high-level optimization policy and the existing compiler becomes a low-level mechanism for applying code transformations. ASSHes are more generic than previous approaches to compiling embedded software: they permit compilation to be directed towards meeting many different goals and combinations of goals.

So far we have investigated ASSHes that control two optimizations: loop unrolling and function inlining. We evaluate ASSHes by applying them to TinyOS sensor network applications, where they have successfully reduced resource usage. For example, for one large application we can either reduce worst-case stack depth by 16%, reduce code size by 3.5%, or reduce the amount of time the processor is active by 14% (our baseline for comparison is the output of the nesC compiler, which itself performs aggressive cross-module inlining). These various improvements conflict: they can be achieved individually but not together—dealing with tradeoffs among resources is central to designing efficient embedded systems.

1. INTRODUCTION

Optimizing compilers try to generate code that is fast and not too large—an appropriate design point for desktop and workstation applications. Embedded software developers, on the other hand, are often not well-served by these compilers: their software must run on platforms with diverse and severe *non-functional requirements*. For example, consider a program that will be run on a processor with a low clock speed, limited battery energy, and little space for code and data—in other words, a typical low-power embedded system. Instead of creating fast and not-too-big code, the compiler should aggressively try to generate code that fits into the available code and data space. Furthermore, the compiler should expend extra optimization effort on tasks with real-time requirements.

Developers using current tools can affect optimization heuristics in various ways that are coarse-grained (e.g., global optimization flags) and fine grained (e.g., profile data, pragmas). In general this control is insufficient for meeting complex application- and platform-specific goals that involve multiple resources. There are two fundamental problems with existing tools:

1. Compilers are oblivious to the high-level non-functional requirements that are externally imposed on an embedded system.

2. Compilers rely on a multitude of built-in heuristics for deciding when to apply optimizations that are not universally beneficial. It is difficult to gain effective overall control over a diverse collection of opaque heuristics.

Lacking compiler support for meeting high-level resource constraints, developers are forced to manually specialize code to tune resource usage. Manual specialization is harmful because it is time-consuming and can introduce bugs. Also, it discourages reuse: the resulting code often sacrifices modularity and, furthermore, it is fragile with respect to changes in the underlying hardware platform: today's resource constraints are not necessarily tomorrow's.

This paper proposes a way to make existing compilers more suitable for compiling embedded software: we use *application- and system-specific heuristics* (ASSHes) to override optimization decisions made by the compiler. ASSHes have access to high-level system requirements and to external program analyzers, code instrumentation techniques, feedback mechanisms, etc. to direct their choices. Particularly important is giving ASSHes access to *resource bounding analyses* [2, 21] that can statically determine if a program will, for example, dynamically run out of memory. In effect, the ASSH becomes the high-level optimization policy while the existing compiler becomes the low-level mechanism for applying code transformations. This mechanism/policy separation is crucial: ASSHes present developers with a very narrow interface to the compiler, giving them the capability to extend it in a constrained way. In contrast, a generally extensible compiler is more powerful but exports a much wider interface, for example exposing the compiler's intermediate representations.

The benefits of ASSHes come not from including any particular analyzer or heuristic—after all, compiler writers could equally well include that analyzer or heuristic—but from the ease with which new ASSHes can be written or combined. This is a corollary to Robison's observation [23] that

> ... small programming niches that would greatly benefit from certain optimizations cannot get those optimizations implemented, because payback for the compiler vendor is not there.

Robison was not talking about the embedded domain but his observation is particularly applicable there. The large diversity of architectures, compilers, and operating systems for embedded systems produces a high degree of market fragmentation. This fragmentation, in combination with the diverse non-functional requirements imposed on embedded systems, guarantees the existence of many small niches whose needs are not met by off-the-shelf tools.

The contributions of this paper are:

- We describe ASSHes: a novel organization for extensible optimization heuristics that exploits existing optimization passes in order to meet application- and platform-specific goals. We intend ASSHes to be developed by programmers with a basic understanding of compilers and compiler optimizations—not by compiler specialists.
- We show that **ASSHes are effective**: we present experimental results showing that they can produce code substantially better than that produced by competing techniques. For example, on a large example we can reduce stack memory consumption by 16% or we can reduce the amount of time that the processor is active (and using energy) by 14%.
- We show that **ASSHes are general**: they support a variety of underlying optimizations (in this paper we describe ASSHes that utilize loop unrolling and function inlining) and they permit a wide variety of high-level goals to be met. ASSHes can also be composed to meet more complex goals.
- We show that **ASSHes are practical**: they can be added to an existing compiler such as gcc either by modifying its code in minor ways, or by preprocessing its input using a source-to-source transformer.

ASSHes significantly generalize our previous work on reducing stack depth [21], which considered tradeoffs between stack memory and code memory.

Section 2 provides a concrete introduction to application- and system-specific heuristics for optimizing embedded software by using a variety of ASSHes to optimize a nesC/TinyOS program. In Section 3 we describe the use of aggressive feedback-driven ASSHes to better compile embedded software. Section 4 describes the interface between ASSHes and the compiler, and how to modify compilers to support ASSHes. In Section 5 we evaluate the effectiveness of several ASSHes in reducing resource usage of a number of TinyOS applications that run on Berkeley motes [10]. We compare ASSHes to previous work in Section 6 and conclude in Section 7.

2. USING ASSHes

This section introduces ASSHes and shows what they can accomplish when applied to a large TinyOS application. This is, in many ways, the key section of the paper: ASSHes described here illustrate all of the important ways in which our approach differs from most previous research. In particular:

- ASSHes can be created by application developers, as opposed to compiler specialists,
- ASSHes can base their decisions on the results of strong resource-bounding analyses,

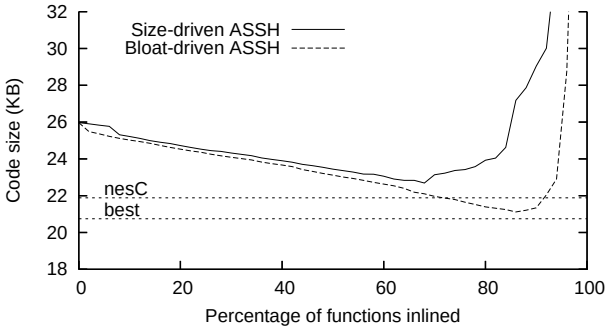

Figure 1: Code size for the `Surge_Reliable` application using two simple ASSHes, nesC, and our best heuristic.

- ASSHes can easily be made aware of high-level platform- and application-specific requirements,
- multiple ASSHes can be composed to support more sophisticated optimizations, and
- ASSHes provide a convenient platform for making use of feedback to improve generated code.

2.1 Background: TinyOS and nesC

TinyOS is extremely lightweight systems software for the Berkeley mote sensor network platform [10]. Motes are microcontroller-based embedded systems equipped with sensors and a packet radio. The need to conserve energy and reduce costs imposes severe resource constraints: the mica2 motes that we use as an experimental platform are based on Atmel's ATmega128 chip running at about 7 MHz. The Atmel chip implements the 8-bit AVR architecture. It has 128 KB of flash memory for storing programs and 4 KB of RAM for data.

The nesC language [8] addresses the particular requirements of the mote platform by providing static component linking, whole-program optimization through inlining, and static detection of concurrency errors. The nesC compiler emits C that is fed to a gcc cross-compiler for AVR. Throughout this paper, our baseline for comparison is provided by the binaries generated by nesC and gcc.

The `Surge_Reliable` application, developed at Crossbow Inc., is the motivating application for this section. It performs multihop routing of sensor data to a sink node. `Surge_Reliable` is relatively large: it uses code from 43 components and, when compiled by nesC, generates about 15,700 lines of C code in about 500 functions.

2.2 Code-size-driven ASSHes

Our first example uses a variety of heuristics to achieve code-size goals using the compiler's function inliner as the underlying optimization mechanism. Since the AVR lacks an instruction cache, we would expect more inlining to be better, as long as code-size limits are not exceeded. In this section we consider coarse-grained inlining decisions that either inline all calls to a particular function, or none of them. In Section 3 we extend our model to cover callsite-specific inlining policies.

Our first two heuristics take a parameter n that indicates what percentage of a sorted list of functions to inline. The first heuristic is very simple: it sorts functions in order of increasing size and then inlines the first n%. The second heuristic uses a more accurate model of code bloat: the cost of inlining a function is estimated to be $(c-1)(s-o)$ where c is the number of static calls to the func-

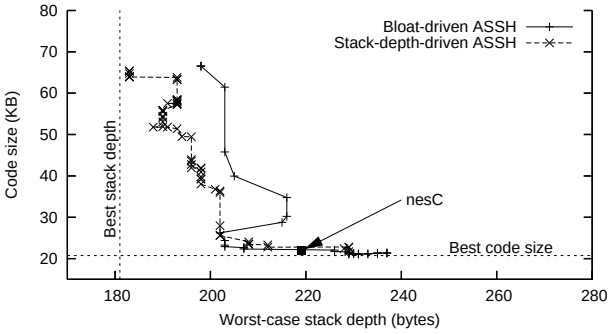

Figure 2: Stack depth vs. code size tradeoffs for the bloat-based inlining ASSH and the stack-depth-driven ASSH described in Section 2.3 applied to the `Surge_Reliable` application. The "best" results are computed by an ASSH that we describe in Section 3.

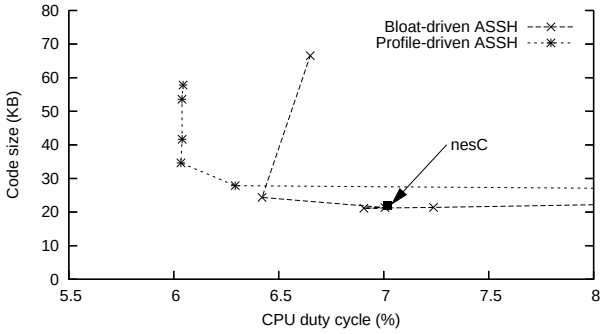

Figure 3: Tradeoffs between code size and duty cycle for the `Surge_Reliable` application using our profile-driven and code-bloat-driven inlining ASSHes. Our best result uses 14% less processor time than does the executable produced by nesC.

tion, s is function size, and o is the minimum function call/return overhead (six bytes on the AVR platform). After deciding to inline a function, the callgraph and the code size of all calling functions must be updated. The results of both heuristics are shown in Figure 1, and they are compared to the size of the executable produced by nesC, and to the code size from our best heuristic. In contrast to the heuristics described in this section, the best heuristic (described in Section 3) is very costly in terms of compile time.

For both of these heuristics, we determine the size of each function by performing a trial compilation with inlining disabled. An alternative is to estimate code size from source code; this is what nesC does. We implemented this, but for obvious reasons it does not work quite as well as the trial compilation.

We also developed two more sophisticated ASSHes for code size. The first uses a heuristic search to find the percentage of functions to inline that minimizes code size; it works well when there are few local minima in the code size vs. $n\%$ function (this is true for a number of examples that we have looked at). The second heuristic uses a binary search: it attempts to maximize the percentage of functions inlined while respecting a given upper limit on code size. This result replicates part of the work described by Leupers and Marwedel [15]—they search more sets of inlining decisions and also incorporate performance feedback from simulator—at an incremental cost of about a dozen lines of Perl. Both of these heuristics make use of *feedback-driven compilation* where the results of trial compilations are used to inform future decisions. We do not present results for the search-based heuristics as they merely serve to find a desirable point on a curve in Figure 1.

2.3 A stack-depth-driven ASSH

TinyOS makes very efficient use of the small amount of RAM present on the motes and, in fact, the only form of dynamic memory allocation in most TinyOS programs is the C call stack. Our previous work [21] created a static analyzer that computed a tight upper bound on the extent of the TinyOS stack, even in the presence of interrupt handlers. The analyzer was based on a context-sensitive dataflow analysis of AVR object code. In that work we also showed that function inlining could be used to significantly reduce the worst-case depth of the call stack by avoiding the calling convention and by facilitating further optimization of inlined code. The optimization was effective but our method was crude, sometimes requiring several hours to optimize a single program.

We now present a novel ASSH for rapidly creating a TinyOS program with small worst-case stack depth. The starting point for this heuristic is the set of inlinings from the previous section that produces the smallest code size. To this set of inlinings we add all function calls that are active when the stack depth exceeds some tunable threshold. Stack depths are measured statically using our analysis tool. The insight behind this heuristic is that systems contain many functions that only operate at low stack depths, and there is little point inlining them. However, functions that are active when the stack is deep should be inlined.

Figure 2 depicts stack depth/code size tradeoffs produced by this new heuristic, by the bloat-based heuristic from the previous section, and by nesC. This graph plots stack depth against code size to make it easier to see the tradeoffs between two resources and to compare the ability of different ASSHes to make different tradeoffs. For a given code size, the stack-depth-based ASSH consistently creates executables with smaller stack depth than does the bloat-based heuristic. We expect that for many TinyOS applications, it will be desirable to reduce RAM consumption at the expense of a code size increase, because the AVR has 32 times more code space than data space. The approximately 40 bytes of stack memory that we save relative to nesC is significant, and would permit the instantiation of a number of additional TinyOS components.

Finally, notice that between these heuristics, we have data points that approximate the best known code size for the `Surge_Reliable` example, and data points that approximate the best known worst-case stack depth. Usage of these two resources cannot be simultaneously minimized (we believe) since they inherently conflict.

2.4 A profile-driven inlining ASSH

To improve the CPU usage (and power consumption) of a TinyOS application, we developed a profile-driven inlining ASSH. Since gcc's profiling support does not work on the mote platform, we modified the cycle-accurate Atemu simulator [20] to produce output equivalent to profiler output.

Our profile driven ASSH is simple: it inlines the $n\%$ most-frequently called functions. To evaluate the speedup generated by an ASSH we compute the *duty cycle* of the CPU on the mote: the time that it spends awake and running (as opposed to asleep and saving power) during a 3.5 billion cycle simulation run (about 500 seconds of simulated time). We found that such a long run was necessary to ensure that we are measuring steady-state results rather than initialization effects: the `Surge_Reliable` application takes a long time to bootstrap into multihop routing mode.

Figure 3 evaluates tradeoffs between duty cycle and code size

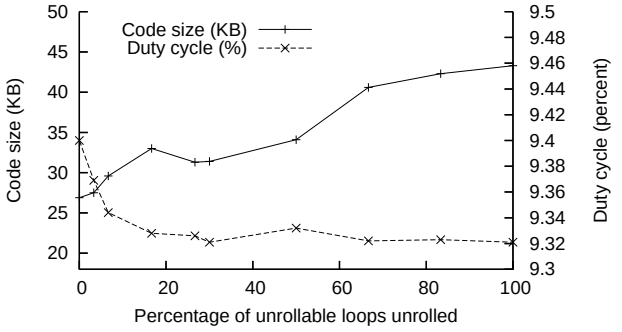

Figure 4: Code size and duty cycle for the `Surge_Reliable` application using our profile-driven unrolling ASSH. Note that the left and right-hand y-axes are different.

space for executables created by our profile-driven ASSH, our code-bloat-driven ASSH, and by nesC. Our best result has a 14% lower duty cycle than does the nesC output. We also measured the duty cycle of executables produced by our stack-depth-driven ASSH—these results were roughly comparable those produced by the code-bloat-driven ASSH and we do not show them here.

The code-bloat-driven ASSH actually generates slower code at the largest code size. We were initially surprised that increased inlining could lead to decreased performance on a processor lacking an instruction cache. On inspecting the output of the compiler we found that when inlining creates very large function bodies, gcc is unable to effectively use PC-relative branch instructions, which on the AVR can only jump 64 instructions in either direction.

Since ATmega128 chips can sleep, consuming little power, when they have no work to do, we expect that our profile-driven ASSH will reduce CPU power consumption by around 14%. If we could reduce the mote CPU's voltage and clock frequency, power savings would be even more dramatic since voltage scaling can yield a proportional reduction in the total power required to execute a program [3]. A limiting factor in applying this approach in embedded systems is that time-sensitive activities may be slowed down so much that the system stops operating properly. We believe that ASSHes can be used to preferentially optimize activities with tight time constraints, in order to create systems that can operate properly at lower clock frequencies than would otherwise be possible. We have not performed this experiment since we cannot change the voltage and frequency on our test boards. However, the compiler side of this experiment can be straightforwardly handled within the ASSH framework.

2.5 A profile-driven unrolling ASSH

Most of the ASSH results presented in this paper are based on function inlining, which we have found to be extremely effective because it has such pervasive effects on the generated code. In this section we explore an ASSH that controls loop unrolling, which helps amortize the cost of the loop termination test and may enable further loop-body optimizations. However, this optimization can also greatly increase code size without yielding any benefit, making it an ideal candidate for ASSHes.

Our profile-driven ASSH unrolls n% of the hottest loops. Figure 4 shows that results of this experiment. This shows the effect of unrolling on code size and on the duty cycle as we increase the percentage of unrollable loops unrolled. No inlining was performed in this experiment. The results are disappointing: there is a definite trend of improvement but the maximum benefit is only 0.8%.

It turns out that the `Surge_Reliable` application contains only 30 unrollable loops, of which only five are executed more than 100,000 times in a 3.5 billion cycle profiling run. Additionally, our unrolling hooks in gcc do not control the degree of peeling and unrolling of some kinds of implicit loops such as those produced by gcc's code generator to perform memory copies and multi-bit shifts. This is a symptom of a general problem in adding ASSH hooks to a compiler: a compiler often has multiple versions of the same optimization because the compiler uses several different intermediate representations. One must take care to add ASSH hooks to each version.

Note to reviewers: We ran out of time to fix this problem. We will modify our experiment to use moderate amounts of inlining, and we will modify gcc to allow our ASSH to control peeling and unrolling in the code generator. We believe these changes will increase the benefit of unrolling from 0.8% to 2–4%.

2.6 WCET-driven ASSHes

The two previous sections describe ASSHes that use dynamic profile data to speed up an application. In this section we use profile-driven function inlining with a major difference: we use *static* profile data from a worst-case execution time estimator. The *worst-case execution time* (WCET) of a block of code is an upper bound on the amount of time that the code may take to execute, regardless of program input data [2, 7]. Determining WCET is obviously undecidable, but in practice good estimates can often be found for embedded software running on simple processors. Embedded software tends to have runtime that is relatively independent of input data, and it also tends to avoid computations with highly dynamic behavior.

We believe that there are two main reasons why it is desirable to perform profile-driven optimization based on static profile data from a WCET analyzer:

1. Acquiring dynamic profile data can be very slow (if cycle-accurate simulators are involved), and simulators require input data and accurate models of I/O devices. WCET, on the other hand, can be estimated quickly and without input data or device models.

2. Reducing WCET can increase the static predictability of a system in the sense that a function that used to have an execution time in the range $[0..C_1]$ will now execute for a time in the range $[0..C_2]$, where $C_2 < C_1$. We believe (but have not shown) that reducing WCET will also increase dynamic predictability. That is, by preferentially optimizing long paths, it will tend to decrease observed variation in execution time. We hope to demonstrate this effect in the future.

We have implemented a WCET estimator for AVR object codes; it reuses the underlying analysis framework from our stack-depth estimator [21]. In some ways our tool is quite sophisticated: it deals well with recursion, irreducible flow graphs, and some forms of indirect calls and jumps. Furthermore, it aggressively avoids analyzing false paths using a context-sensitive dataflow analysis. In other respects, our tool is primitive: it makes pessimistic assumptions about instructions with variable execution times and it requires human intervention to bound loop iterations. We omit further details about our tool, as it is largely based on known techniques. Furthermore, as far as the ASSH described in this section is concerned, the details of the underlying WCET tool are not very important: should a better tool become available, we could start using it with our ASSHes right away.

A useful by-product of WCET analysis is a worst case execution count (WCEC) for each instruction. We use WCEC just like profile

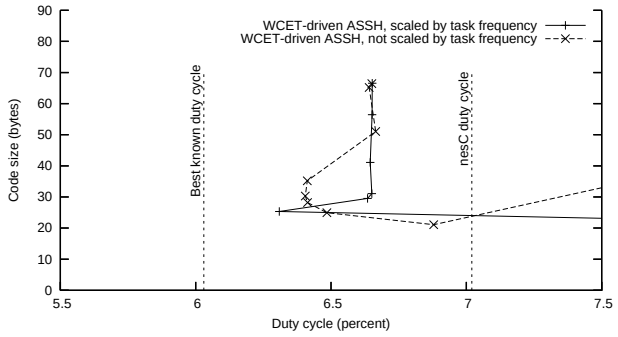

Figure 5: Duty cycle vs. code size tradeoffs for our two WCET-driven ASSHes

data: to inline the hottest n% of functions. The only significant obstacle to doing this is the fact that the WCECs that we compute are per invocation of a task or interrupt. To compute WCEC per time unit, we need to multiply analyzed WCEC by invocation frequency—the number of times per second the task or interrupt is signaled. It seems unlikely that the invocation frequencies can be found through static analysis, so we must get this information from another source, such as a specification or a single run of a simulator.

Figure 5 shows code size vs. duty cycle tradeoffs for two ASSHes: one that scales the WCEC of each function by the observed invocation frequency of its enclosing task or interrupt, and one that does not. As expected, both WCET-driven ASSHes perform more poorly than the profile-driven ASSH. Also as expected, the scaled WCECs produce better code, but by only about 1.5%. We believe that WCET (and WCEC) can play an important role in optimizing embedded software, and we hope to revisit this subject in more depth in the future.

2.7 Composing ASSHes

Our composition strategy for ASSHes is simple: they combine at the level of individual optimization decisions using basic set operations. For example, our stack-depth-driven ASSH starts with a set of inlining decisions that has been shown to result in good code size, and then adds another set that is believed will either improve stack usage. Clearly this strategy requires careful use: it will not work when ASSHes conflict by canceling each others' effects. When dealing with very expensive optimizations, set intersection might be a better operator than set union: an optimization is only applied when multiple ASSHes all believe it will be beneficial.

In a few cases we have combined ASSHes based on set subtraction: some functions that should never have loops unrolled and should never be inlined. For example, TinyOS contains a calibrated delay loop `TOSH_uwait`—unrolling the delay loop is worse than useless as it bloats code size while decalibrating the loop. In some cases we want to avoid inlining the `TOS_post` function, which makes an event handler runnable: this is because our static analyzer heuristically relies on finding calls to this function in order to infer which functions serve as asynchronous TinyOS tasks.

We believe that it will sometimes be beneficial for ASSHes to interact at higher levels of abstraction, for example by sharing information about the expected effects of optimizations that they have selected. We leave this for future work.

2.8 Summary

We have sketched the use of a number of different ASSHes to reduce code size, stack size, and execution time. These ASSHes exercise global control over inlining and/or loop unrolling optimizations. Many ASSHes rely on feedback from trial compilations, offsetting problems caused by the fact that complex optimizations result in unpredictable costs and benefits. Furthermore, many ASSHes make use of strong whole-program analyses such as those that bound execution time or stack memory consumption.

3. AGGRESSIVE USE OF FEEDBACK

When an ASSH author can make effective predictions about the effects of performing optimizations, feedback through trial compilations is unnecessary. However, several of the ASSHes that we described in Section 2 required a parameter whose value would be hard to determine a priori. In these cases, *feedback* is useful: we find a good value of the parameter using trial compilations. For examples we already presented, a few iterations were sufficient.

This section describes a much more aggressive application of feedback: we use it to greedily explore very large search spaces. Although the resulting ASSH is slow, taking up to 12 hours to optimize the `Surge_Reliable` application, this approach is useful in two ways:

1. It provides the best possible overall compilations of embedded systems, based on techniques currently available to us. Therefore it is a good basis for comparison for the faster heuristics. For example, a fast ASSH that provides 95% of the benefit of the slow feedback heuristic is probably quite useful, while a fast ASSH that provides only 30% of the benefit may not be worthwhile.

2. Embedded system developers are often willing to invest time in a final, highly optimized build of a system for actual deployment. For example, consider the extreme case where people developing a digital camera or a cell phone create a new ASIC (application-specific integrated circuit) to handle, for example, image processing or voice processing. The cost of this "optimization" is enormous: it requires skilled staff, a lengthy design cycle, and on the order of half a million dollars for fabrication. Compared to these costs, an overnight compilation is relatively affordable.

Our slow feedback heuristic extends the one that we described in previous work [21] by supporting more resources, by providing a more general framework for searches, and by making an even more aggressive search of the optimization space. It attempts to minimize a *cost function* that describes how to trade resources against each other. For example, one useful cost function might trade 10 bytes of code memory for one byte of data memory for 100 cycles of CPU usage. Other cost functions can include discontinuities to indicate hard constraints. At a high level, one might want to treat code memory and data memory as having zero cost until their respective limits are exceeded, and then to treat their cost as infinite. This cost function, while accurate, will not work in practice: it does not guide the heuristic through the search space to a region containing good solutions. A better function would impose a small cost per byte of code and data memory used, plus a large constant cost once the resource limit is exceeded. In summary, a number of useful application- and system-specific resource metrics can be encoded using cost functions, and we believe that they provide a useful framework for reasoning about tradeoffs among resources.

Our slow feedback heuristic operates as follows:

1. Create an initial list of optimization decisions.

2. Perturb the set of optimizations and see if this improves the cost function. If it does, accept the new set, otherwise back off to the old set.

	stack depth (bytes)		code size (KB)		WCET (total cycles)	
nesC	219		22		59742	
stack, WCET, code	181	(17%)	32	(-45%)	54383	(8.9%)
stack, code, WCET	181	(17%)	24	(-7.4%)	58992	(1.3%)
code, stack, WCET	214	(2%)	21	(5.2%)	60419	(-1.1%)
code, WCET, stack	214	(2%)	21	(5.2%)	60419	(-1.1%)
WCET, stack, code	181	(17%)	36	(-61%)	54361	(9.0%)
WCET, code, stack	185	(16%)	32	(-43%)	54383	(9.0%)

Figure 6: Results of running our slow feedback heuristic with six different cost functions, each with a different prioritization of resources to reduce. Parenthesized percentages indicate improvement relative to nesC.

3. Go back to step 2 until a fixpoint is reached. This can either be a real fixpoint where no further permutations improve the cost function, or a probabilistic fixpoint where none of the last n permutations has resulted in an improvement.

Applying this heuristic to function inlining is straightforward. The initial set of inlining decisions is empty, the permutation functions adds or subtracts a configurable number of inlining decisions at each iteration, and the fixpoint can be either probabilistic or deterministic. The slow feedback-driven inlining ASSH can be configured to inline all calls to a function, or to inline individual callgraph edges. In both cases the search space presented by function inlining is large: 2^n where n is the number of functions or the number of callgraph edges.

Note to reviewers: We did not have a chance to try feedback ASSHes that control unrolling before the submission deadline. We will do this for the final paper—it should be straightforward.

There are six ways to prioritize the reduction of three resources. Figure 6 shows the results of applying these six prioritizations to the reduction of stack depth, code size, and total worst-case execution time of the `Surge_Reliable` application. (Total worst-case execution time just adds up the WCET of each task and interrupt handler in an application.) Note that stack depth and WCET can be simultaneously driven to near their individual minimum values, while optimizing for code size appears to directly conflict with both reduced stack depth and reduced WCET.

Since inlining permits a function to be specialized for its calling context, it interacts strongly with other optimizations such as register allocation, constant propagation, and dead code elimination. In effect, inlining serves as a meta-optimization that controls the amount of code that other optimizations can see at once. Since the interactions between these optimizations are complex, inlining can be highly unpredictable. Consider the fact that (ignoring cases where usage of one or more resources is unchanged) there are eight possible effects that an inlining decisions can have: code size can increase or decrease, stack depth can increase or decrease, and worst-case execution time can increase or decrease. We looked at the results of a number of inlining decisions, and found that all eight possibilities actually occurred while running our slow feedback heuristic on the `Surge_Reliable` example. Certainly it was surprising at first to see inlining cause stack depth or execution time to increase, but on reflection it is not hard to find mechanisms explaining these effects.

4. THE ASSH INTERFACE

When a compiler has to decide whether to apply an optimization or not, it usually invokes a heuristic to make the decision. Typically, each such heuristic has a different interface. For example, it may be passed a piece of intermediate code and return a yes/no answer, or it may be passed a loop body and return a decision whether to unroll it n times, peel it n times, or do neither. Naturally this interface is specific to individual compilers, and even to different versions of the same compiler. The purpose of the ASSH interface, described in this section, is to smooth over these differences and expose opportunities for optimizations to external tools. Naturally, an ASSH will be required to make its decisions using less compiler-internal information than is available to the built-in heuristics. Even so, we believe that ASSHes can often outperform built-in compiler heuristics because they can make use of high-level information about applications and platforms, resource-bounding analyses, and information about high-level application- and platform-specific requirements In cases where ASSHes do not perform well relative to built-in compiler heuristics, our fallback position is simple: the ASSH should defer to the existing heuristic.

4.1 ASSH interface design principles

ASSHes communicate with the compiler through a deliberately narrow interface that follows a few basic design principles:

Separate mechanism and policy — Compilers do a poor job of separating optimizations from policies for deciding when to apply the optimizations. Our first-order design goal is to provide high-level policies with a clean interface to the low-level mechanisms.

Usable by developers — One of our goals is for normal embedded system developers, as opposed to compiler specialists, to be able to create new ASSHes. This means that ASSHes should be primarily about the program being compiled, rather than being about the compiler. Intermediate representations should not be exposed to the ASSH writer, and pieces of code should be identified by their source location—not some internal loop number, basic block number, or function identifier. Although this paper does not demonstrate portability of ASSHes to compilers other than gcc, we believe they could be ported with modest effort.

Minimal — ASSHes could potentially be supplied with the same set of information that compiler-internal heuristics are given. We opt for a more constrained interface in order to avoid creating ASSHes that are dependent on a compiler or even a compiler version. Also, we expect that ASSHes will primarily get information from high-level resource requirements, from external analysis tools, and from feedback. Of course, given a compelling reason, the interface to any individual underlying optimization could be extended to include more information.

Offline — In our early experiments, ASSHes were executed *online*: their execution was interleaved with that of the compiler (they were linked in as dynamically loaded libraries). We hoped that permitting ASSHes to make incremental decisions would allow them to arrive at better global solutions. This turned out not to be the case: the compiler requested optimization decisions in (effectively) random order, making it difficult to exploit information from previous decisions. Our current offline interface presents the compiler with a set of decisions stored in a file. This makes ASSHes easier to develop (they can be in any language) and makes their effects easier to track, since all decisions pass through the filesystem.

4.2 The ASSH interface

Figure 7 shows an ASSH in its larger context. The ASSH interface that we have developed works as follows:

1. A list of sites to potentially optimize is created. This list could come from a trial compilation, from an external tool, or could even be generated by manual inspection of the program.

2. The ASSH decides which sites to optimize (and, if appropri-

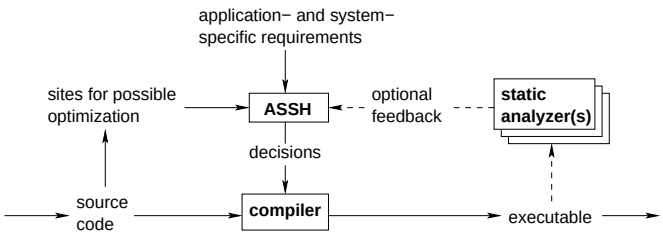

Figure 7: An ASSH in context

ASSH mechanism	inputs	output
inline	srcloc of caller srcloc of callee	true / false
unroll	srcloc of loop start srcloc of loop end	N copies

Figure 8: The ASSH interface to two compiler optimizations

ate, to what degree), which sites not to optimize, and which to leave to the compiler's built-in heuristics.

3. The compiler creates object code conforming to the ASSH's decisions.

4. Optionally, go back to step 2 after examining the generated code. We discuss use of feedback in more detail in Section 3.

At present, we have added ASSHes to two compiler passes using the interfaces shown in Figure 8. A "srcloc" serves as a way to track a particular piece of code as it passes through the compiler.

4.3 Making compilers listen to ASSHes

In this section we describe the two ways that we made gcc perform the optimizations specified by ASSHes. At a high level, the mechanism used to make a compiler respect the decisions made by ASSHes is irrelevant to the thesis of our paper. However, practically speaking, implementation cost and impact on compiler structure must be modest.

The first way to override compiler optimization heuristics is to modify the compiler. In our experience, this is fairly easy. One must identify the existing heuristic, modify it to consult the ASSH decision list, and then check for validity of the transformation. In practice, it is often desirable to run the built-in heuristic as well as consulting the ASSH decision. This makes the compiler patch less invasive and also permits the compiler to operate as before if the ASSH has no particular preference.

We used this approach to add ASSH control to the loop unrolling optimizations in gcc. gcc performs two related loop optimizations: it may *peel* a loop by inserting one or more copies of the loop body before the loop (reducing the number of loop iterations by the number of copies made); and it may *unroll* a loop by inserting one or more additional copies of the loop body inside the loop (dividing the number of loop iterations by the number of copies of the body inside the loop). In many cases, it must use a combination of the two: if a loop executes 103 times, one might peel three iterations and unroll 10 times yielding code that is up to 13 times larger but iterating only 10 times. Where no confusion arises, we use "loop unrolling" to refer to both optimizations.

gcc version 3.4 has two clearly-identified sites where it decides whether and how much to unroll loops. Both sites that make loop unrolling decisions have a very clear separation between the heuristics that make decisions and the code that implements the decisions, so it was straightforward to insert code to consult tables of optimization decisions loaded from a file. While experimenting with ASSHes we found it convenient to perform the lookup by invoking a function dynamically linked with gcc. We finally settled on two versions of this function: one simply prints its arguments and is used to generate a list of sites for possible optimization and the other performs a lookup. The gcc changes required to control loop unrolling consist of 209 new lines of code and no changes to existing lines of code. (This number excludes $333 + 20$ lines of changes required to track source locations accurately: described in Section 4.4.)

While it was easy to add support for ASSHes, it was more difficult to find a good ASSH interface. In all, gcc recognizes five different forms of peeling and unrolling and each is parameterized by the number of times to peel or unroll. In early versions, we required ASSHes to specify which of these five optimizations to apply. This proved to be complex to use and, since each optimization has its own preconditions, either required ASSHes to know which optimizations could be chosen for each loop or else required gcc to convert an ASSH's request into the "next best" valid choice. After some experimentation, we settled on a significantly simpler interface: the ASSH only specifies the maximum permissible number of copies of the loop body. If only one copy is allowed, no optimization is performed; if 15 copies are allowed, a loop that executes exactly 103 times might be peeled three times and unrolled 10 times (yielding 13 copies).

The second way to override compiler optimization heuristics is to perform a source-to-source transformation that either does the optimization or somehow enables the C compiler to do it. To support inlining ASSHes we built on our previous work on cross-module inlining [22]. The approach is essentially that of Cooper et al. [5]: we parse a collection of preprocessed C compilation units, perform appropriate renaming and unification of data types, topologically sort functions in order to satisfy as many of the requested inlinings as possible, and then output a new C file with some functions marked as inline. Furthermore, we perform *caller-specific inlining*—inlining some calls to a function but not others—by emitting two copies of a function, one marked inline and one not, and selectively transforming call-sites to call one or the other. Finally, the generated file is compiled by gcc with a carefully selected set of command-line options indicating that it should obey the requested inlining decisions, but not make any on its own.

This approach works, but it has been painful. First, implementing and maintaining a gcc-compatible parser and typechecker is a lot of work (had CIL [18] been available when we started our work, it would have been easier). Second, some recent versions of gcc implement inlining behavior that cannot be overridden with any set of command-line options. We are forced to modify these compilers (changing one or two lines suffices) to defeat this behavior. Overall, several years of experience with the preprocessor approach has caused us to agree with Robison [23], who gives a number of solid reasons for avoiding it.

4.4 The code location problem

Some ASSHes need to relate instructions in the object code to the optimizations that produced them. For the most part, we can rely on existing infrastructure: many compilers routinely track source code locations across various optimization passes and into generated object code. This infrastructure does not work well when code duplication occurs, for example through loop unrolling and function inlining. The basic problem is that if ASSHes' optimization

requests are phrased in terms of source-code locations, then it is not possible to specify which of the resulting copies should be optimized.

An analogous problem occurs when an object code analyzer requires source code annotations. For example, our WCET analyzer must map loop locations in the source code to locations in the object code in order to take advantage of user-provided loop iteration bounds. Our original worst-case execution time analyzer overcounted costs when a loop was unrolled: the analyzer used the original (unrolled) loop iteration bound instead of scaling the loop bound by the number of times the loop was unrolled. Schnayder et al. [24] report a similar problem with PowerTOSSIM, a simulator that estimates the power consumption of a TinyOS program running on a Berkeley mote. PowerTOSSIM estimates the power consumption of basic blocks identified in the source code by estimating the cost of all the corresponding instructions in the object code, causing them to overcount the cost of each basic block by (as a first approximation) the number of copies made.

Jaramillo et al. [11] developed an elegant solution to the source mapping problem in the context of debugging optimization passes. Their key idea was to label each static instance of a loop body with triples of the form $\{l, u, i\}$ indicating that a particular instance would execute iterations l, $l + i$, $l + 2i$, $\ldots u$ of the original loop. For example, the body of a loop in the source code is (implicitly) labeled $\{0, last, 1\}$ indicating that it is used for iterations 0, 1, $\ldots$ of the loop. If the loop is unrolled, this label is used to generate a new label for each instance of the loop body. For example, if a loop is unrolled two times, then the instances would be labeled $\{0, last, 1\}$ and $\{1, last, 2\}$ indicating that they are used to execute iterations 0, 2, 4, $\ldots$ and 1, 3, 5, $\ldots$, respectively. Jaramillo et al.'s labels are closed under many common loop optimizations: as loop optimizations are applied, the labels on the originals can be used to determine the new labels.

Jaramillo et al.'s approach does not handle function inlining but it is easily adapted to do so using the same idea of identifying the potential dynamic instances of the function that could exist at runtime. Consider the following program (we have included line numbers to let us reference source code locations in the following):

```
1    void bar() {
2      x = 1;
3    }
4    void baz() {
5      bar();
6      bar();
7    }
```

If we inline both calls to `bar`, we will end up with two instances of line 2 and the source locations would normally change as follows:

```
4    void baz() {
2      x = 1;
2      x = 1;
7    }
```

Any reference to line 2 would now be ambiguous. The solution is to concatenate source locations of the call site with the locations within the callee, as follows:

```
4    void baz() {
5.2    x = 1;
6.2    x = 1;
7    }
```

and we can now refer unambiguously to each instance of line 2. If line 2 contained a function call or a loop, we would be able to direct optimization to instance 5.2, instance 6.2, or to both (which may be written $*.2$ or, more conveniently, just 2).

Transforming source locations in this way also solves the problem for our WCET analyzer because it allows the analyzer to compute how many times each loop instance is executed if the original loop executed some known number of times. Each loop instance $\{l, u, i\}$ in the object code corresponds to dynamic instances l, $l+i$, $l + 2i$, $\ldots u$ of the original loop. Therefore, if the original loop executes N times, the number of times each loop instance executes is the number of non-negative, integral values of k such that:

$$0 \leq l + ki < \min(N, u)$$

Implementing our extension of Jaramillo et al.'s labels was straightforward if tedious. Since gcc already labels every instruction with a source location and carefully propagates these labels every time instructions are duplicated, merged, or transformed, we chose to implement labels by modifying the filename component of source locations. After gaining familiarity with gcc, it was easy to identify those parts of gcc which copied instructions and to transform the filenames appropriately; the trickiest part was determining which label to apply to each freshly created instance. Implementing this required considerable effort to be spent learning how gcc's loop unrolling and inlining operate and identifying the right places to modify the code. It required 333 new lines of code and 20 changes to existing lines of code. The good news is that this work does not need to be repeated over and over again as new ASSHes are implemented.

5. EVALUATION

So far we have shown that a number of ASSHes can provide good results for a single application: the TinyOS `Surge_Reliable` kernel. Figure 9 shows the results of three of our ASSHes when applied to five other TinyOS applications. As before, our baseline is provided by executables produced by the default nesC/gcc toolchain. Note that the time to run the profile-driven ASSH does not include the time to generate profile data using the simulator. Getting profile data from Atemu can be very fast, or it can be very slow (up to 30 minutes in our experience) when a complicated network of motes needs to be simulated for billions of cycles.

Our results indicate that nesC already does quite well in producing executables with small code size, but that it can be improved upon significantly with respect to stack memory usage and processor utilization. However, these improvements increase compilation time, sometimes significantly. All times were taken on a 2.4 GHz AMD Athlon 64. In the future we plan to investigate ways to improve the speed of ASSHes, perhaps using machine-learning techniques to avoid trial compilations by more effectively predicting the effects of optimizations.

6. RELATED WORK

Our work builds on a large amount of previous research. Here we divide the existing work into several categories and contrast our approach with some representative examples.

Using feedback in compilation. Profile data can be used to guide a variety of optimizations [4, 19]. Several projects have used feedback to solve the phase-ordering problem for optimization passes [1, 13, 14]. Dean and Chambers [6] developed inlining trials, which examine the output of the optimizer after inlining a function call in order to make better inlining decisions in the future. Leupers and Marwedel [15] describe a technique for finding inlining decisions that maximize speedup while remaining within a

ASSH	Bloat-driven				Stack-depth-driven				Profile-driven			
Resource metric	code size (bytes)				worst-case stack depth (bytes)				average duty cycle (%)			
	ASSH	nesC	imp.	time	ASSH	nesC	imp.	time	ASSH	nesC	imp.	time
`Blink`	1612	1620	0.49%	5 s	33	40	17.5%	5 s	0.030%	0.031%	4.8%	1 s
`CntToLedsAndRfm`	10512	10886	3.44%	29 s	97	128	24.2%	41 s	5.55%	6.28%	12%	3 s
`Oscilloscope`	6730	6960	3.30%	75 s	83	118	29.7%	24 s	0.43%	0.43%	0.0%	3 s
`RfmToLeds`	9898	10300	3.90%	27 s	97	128	24.2%	37 s	5.14%	5.61%	8.4%	3 s
`Surge_Reliable`	21622	22406	3.50%	75 s	183	219	16.4%	113 s	6.03%	7.02%	14%	5 s
`TOSBase`	10364	10774	3.81%	18 s	97	128	24.2%	36 s	5.27%	5.77%	8.7%	2 s

Figure 9: Evaluation of three ASSHes applied to six TinyOS kernels. The "imp." columns indicate improvement in resource usage of the ASSH vs. nesC. The "time" columns indicate the total time to compile the application using the ASSH.

code size budget. They used a branch-and-bound algorithm to limit the number of trial compilations, and evaluated the performance of each trial using a cycle-accurate simulator.

Zhao et al. [29] describe an optimization framework based on accurate models of caches and of loop optimizations which affect memory access patterns. Wang [27] describes a mechanism for efficiently modeling the performance of superscalar architecture inside a compiler using symbolic representations to handle unknowns. Our WCET-based ASSH also uses a model to make optimization decisions, but the model is somewhat different and it is provided by an external tool rather than by the compiler. On the other hand, our feedback-based ASSHes represent a completely different approach: the model is replaced with data about the actual system being compiled, potentially improving accuracy at the expense of increased compile time.

There is little previous work on using *static* profile data to drive optimization decisions. Zhao et al. [30] is the only work that we are aware of that optimizes using feedback from a WCET analyzer. Our work extends Zhao et al.'s by optimizing an entire working system instead of attacking isolated benchmark programs, by comparing speedups that can be achieved using static vs. dynamic profile data, and by exploring the role of task execution frequencies in creating good static profile data.

All of these previous efforts, except Zhao et al.'s work on reducing WCET, have focused on traditional compiler metrics: code size and speed. In contrast, our work makes it possible for optimizers to make use of high-level application- and platform-specific information, and to direct optimizations towards fairly generic goals.

Compiling embedded software. Much research during the past few years has addressed specific problems in compiling embedded software. Broadly speaking, these projects can be divided into those proposing new compiler optimizations and those proposing ways to make more effective use of existing optimizations. The second group of projects is more closely related to our work; here we describe a few representative examples. Again, ASSHes differ from previous work by supporting more generic goals, and by making it easier to incorporate high-level requirements into compilation.

Zhou and Conte [31] showed how to maximize the speed of a program for a given code size budget, and also how to minimize code size without regard for speed. Naik and Palsberg [17] cast register allocation and code generation as integer linear programming (ILP) problems in order to generate code that fits into a given budget. The nesC compiler [8] exploits an existing function inliner in order to generate fast, small code. In the commercial world, the CodeBalance tool from Green Hills [9] is the only tool we are aware of that permits fine-tuning of speed vs. size tradeoffs. Commercial tools are highly constrained by the requirement to avoid increasing compile time: something that we have chosen to sacrifice in many cases.

Extensible compilers. ASSHes are very constrained compiler extensions. They differ significantly from other extensible compilation approaches that have been developed. For example, the Broadway compiler [16] permits sophisticated users to help the compiler make optimizations based on domain-specific characteristics of interfaces. ASSHes, on the other hand, are not at all concerned with program semantics, and they make use of existing optimization passes rather than providing new ones. Extensible compiler frameworks such as SUIF [28] and Soot [25] are significantly different from our work because they expose their intermediate representations. They are designed to be extensible by compiler specialists—not by application developers. Finally, there are mechanisms that permit language extensions to be implemented within the programming language, such as templates [26] and powerful macro systems [12]. These give developers local control over code generation but they are not generally helpful in specializing programs to meet global resource requirements.

7. CONCLUSIONS

Embedded software is very malleable while it is being compiled: the compiler has a great many choices about what transformations to make, and each of these choices potentially affects the usage of multiple resources. Existing compilers are ineffective at exploiting this malleability and will rigidly create, for example, an executable requiring slightly more RAM than the underlying platform supports.

We have presented ASSHes: application- and system-specific heuristics for compiling embedded software. ASSHes represent a step towards our long-term vision of aggressive compiler support for meeting non-functional constraints. We believe that ASSHes represent a sweet spot in the design space for extensible compilers: they permit compilers to indirectly become aware of resource constraints, but do not require extensive modifications to existing compilers, or expose compiler-internal representations. In the long run we believe that ASSHes, or their successors, will provide even more benefit by performing very high-level optimizations such as component and algorithm selection.

8. REFERENCES

[1] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proc. of the*

2004 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES), Washington, DC, June 2004.

[2] D. Brylow and J. Palsberg. Deadline analysis of interrupt driven software. In *Proc. of the 11th Intl. Symp. on the Foundations of Software Engineering (FSE)*, pages 198–207, Helsinki, Finland, Sept. 2003.

[3] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, Apr. 1992.

[4] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic compiler code optimizations. *Software—Practice and Experience*, 21(12):1301–1321, Dec. 1991.

[5] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[6] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proc. of the Conf. on LISP and Functional Programming*, pages 273–282, Orlando, FL, June 1994.

[7] J. Engblom, A. Ermedahl, M. Nolin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Journal of Software Tool and Transfer Technology (STTT)*, 4(4):437–455, Aug. 2003.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[9] Green Hills Software. The CodeBalance code optimizer, 1998. `http://www.ghs.com/download/datasheets/CodeBalance.pdf`.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, Nov. 2000.

[11] C. Jaramillo, R. Gupta, and M. L. Soffa. Capturing the effects of code improving transformations. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 118–123, Paris, France, Oct. 1998.

[12] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proc. of the Conf. on LISP and Functional Programming*, pages 151–161, Cambridge, MA, Aug. 1986.

[13] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 171–182, 2004.

[14] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. 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.

[15] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 253–256, San Jose, CA, Nov. 1999.

[16] C. Lin and S. Z. Guyer. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proc. of the IEEE*, 93(2), 2004.

[17] M. Naik and J. Palsberg. Compiling with code-size constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):163–181, Feb. 2004.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, Apr. 2002.

[19] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 16–27, White Plains, NY, June 1990.

[20] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir. ATEMU: A fine-grained sensor network simulator. In *Proc. of the 1st IEEE Intl. Conf. on Sensor and Ad Hoc Communication Networks*, Santa Clara, CA, Oct. 2004.

[21] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, Oct. 2003.

[22] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. 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, Oct. 2000. Springer Verlag.

[23] A. 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.

[24] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the Second ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, Nov. 2004.

[25] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot—a Java optimization framework. In *Proc. of the CASCON Conf.*, pages 125–135, Toronto, Canada, 1999.

[26] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[27] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 73–84, Orlando, FL, June 1994.

[28] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.

[29] M. Zhao, B. Childers, and M. L. 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.

[30] W. Zhao, P. Kulkarni, D. Whalley, C. Healy, F. Mueller, and G.-R. Uh. Tuning the WCET of embedded applications. In *Proc. of the 10th IEEE Real-Time Technology and Applications Symp. (RTAS)*, pages 472–481, Toronto, Canada, May 2004.

[31] H. Zhou and T. M. Conte. Code size efficiency in global scheduling for ILP processors. In *Proc. of the 6th Annual Workshop on the Interaction between Compilers and Computer Architectures*, Cambridge, MA, Feb. 2002.