Eliminating the Call Stack to Save RAM

Xuejun Yang
University of Utah
jxyang@cs.utah.edu

Nathan Cooprider
BAE Systems AIT
nathan.cooprider@baesystems.com

John Regehr
University of Utah
regehr@cs.utah.edu

Abstract

Most programming languages support a call stack in the programming model and also in the runtime system. We show that for applications targeting low-power embedded microcontrollers (MCUs), RAM usage can be significantly decreased by partially or completely eliminating the runtime callstack. We present *flattening*, a transformation that absorbs a function into its caller, replacing function invocations and returns with jumps. Unlike inlining, flattening does not duplicate the bodies of functions that have multiple callsites. Applied aggressively, flattening results in *stack elimination*. Flattening is most useful in conjunction with a *lifting* transformation that moves global variables into a local scope.

Flattening and lifting can save RAM. However, even more benefit can be obtained by adapting the compiler to cope with properties of flattened code. First, we show that flattening adds false paths that confuse a standard live variables analysis. The resulting problems can be mitigated by breaking spurious live-range conflicts between variables using information from the unflattened callgraph. Second, we show that the impact of high register pressure due to flattened and lifted code, and consequent spills out of the register allocator, can be mitigated by improving a compiler's stack layout optimizations. We have implemented both of these improvements in GCC, and have implemented flattening and lifting as source-to-source transformations. On a collection of applications for the AVR family of 8-bit MCUs, we show that total RAM usage can be reduced by 20% by compiling flattened and lifted programs with our improved GCC.

Categories and Subject Descriptors C.3 [*Special-purpose and Application-based Systems*]: Real-time and Embedded Systems; D.3.4 [*Programming Languages*]: Processors—optimization

General Terms Performance, Languages

Keywords sensor networks, embedded software, compiler optimization, memory optimizations, memory allocation, stack liveness analysis

1. Introduction

Microcontroller units (MCUs) are inexpensive systems-on-chip that are equipped with a limited amount of on-chip RAM: typically between hundreds of bytes and tens of kilobytes. Regardless of whether an MCU is programmed in assembly, C, C++, or Java, it is likely to use some of its RAM to store one or more call stacks: chains of activation records representing the state of executing functions. From the point of view of the compiler, targeting a runtime system based on a callstack has important benefits. First, the stack is memory-efficient: variables live only as long as they are needed. Second, the stack is time-efficient, supporting allocation and deallocation in a handful of clock cycles. Third, compilers can rapidly generate high-quality code by compiling one function at a time; the small size of the typical function makes it possible to run aggressive intraprocedural optimizations while also achieving short compile times.

Our thesis is that despite these well-known benefits: *Partially or completely eliminating the runtime call stack—using our* flattening *and* lifting *transformations—can significantly reduce the RAM requirements of applications running on embedded microcontrollers. Additional savings can be obtained by modifying the compiler to improve its ability to optimize flattened code.*

MCUs typically use a Harvard architecture where code is placed in flash memory and data is placed in SRAM. Unlike flash memory, which only consumes power when it is being actively used, SRAM draws power all the time. Thus, for battery-powered applications that leave the processor idle much of the time, overall energy use can be dominated by the power required to maintain data in SRAM. Popular members of the AVR family of low-power MCUs, which we use to evaluate our work, provide 32 times more flash memory than SRAM.

Reducing the RAM used by an embedded application serves two purposes. First, it can enable an application to fit onto a smaller, cheaper, lower-power MCU than it otherwise would. Second, it can enable more functionality to be placed onto a given MCU.

Our work targets legacy C code and does not require any changes to the programming model. Also, we propose replacing the stack with static memory allocation, not replacing stack allocation with heap allocation (as some functional language runtimes do). In fact, the MCU applications that we are concerned with do not use a heap at all.

Our lifting transformation moves a global variable into `main`'s local scope, and our flattening transformation absorbs a function into its caller, replacing function call and return operations with jumps. Unlike inlining, flattening does not make a new copy of the function's body for each callsite. In the absence of flattening hazards—which are similar to inlining hazards and include recursive calls, indirect calls, external calls, and interrupt handlers—a program can be completely flattened. A completely flattened program has a single stack frame—for `main`—which is a degenerate case since it is allocated at boot time and is never deallocated. Because programs for MCUs typically make limited use of recursive and indirect calls [Engblom 1999], flattening can reduce most programs to `main`, some interrupt handlers, and a few external library calls.

LCTES'09, June 19–20, 2009, Dublin, Ireland.

Reprinted from LCTES'09,, [Unknown Proceedings], June 19–20, 2009, Dublin, Ireland., pp. 1–10.

UU IR Author Manuscript

UU IR Author Manuscript

Aggressive flattening saves RAM in the following ways:

- The function calling convention is bypassed, avoiding the need to push data such as the return address, frame pointer, and caller/callee-saved registers onto the stack. It is still necessary to track the return site of each call, but this information is stored in regular scalar variables and is therefore exposed to the register allocator and other optimizers.
- Compilers whose intraprocedural optimizers are stronger than their interprocedural optimizers (i.e., all compilers) can emit better code from flattened functions by considering more code at once.
- Lifting variables into `main`'s scope exposes them to more aggressive optimizations.

In effect, flattening and lifting trick a legacy C compiler into performing whole-program optimizations. This works because modern compilers, running on modern desktop machines, are perfectly capable of whole-program compilation for MCU-sized codes.

Stack elimination is not without risks. First, we have found that flattening increases application code size due to insertion of function return tables and explicit initializers for lifted global variables. Second, it is possible that aggressive flattening can increase register pressure to the point where many spills occur, defeating RAM savings. Even so, modern versions of GCC reduce the RAM usage of flattened code for almost all applications that we tested. Furthermore, we implemented several improvements to GCC's AVR port, and show that these result in additional RAM savings. Our improvements are in live range analysis and in stack frame allocation.

Our principal result is that we can reduce total RAM usage of a collection of embedded applications by an average of 20%. Total RAM usage is measured by adding the amount of statically allocated RAM (in the data and BSS segments) to the worst-case size of the callstack, if any. The drawback is that code size is increased by 14%. Thus, while flattening is not applicable to all programs, it represents a useful alternative for RAM-limited programs on embedded architectures that provide plenty of code memory. The CPU usage of our test applications was not greatly affected; they are 3% faster, on average, after flattening.

Stack elimination has two interesting side effects. First, since function return addresses are no longer stored on the stack, programs become less vulnerable to control-flow hijacking via buffer overflow vulnerabilities. Recent work has shown that even programs for Harvard-architecture MCUs may be vulnerable to code injection attacks [Francillon and Castelluccia 2008]. Flattening does not simply provide security through obscurity; we argue in Section 5.2 that we have developed a new, viable implementation of control flow integrity [Abadi et al. 2005]. The second useful side effect of stack elimination is that the callstack is a source of unpredictability in MCU software: developers are often uncertain about its maximum extent. After stack elimination, computing the worst-case RAM requirement of a program is trivial.

2. Flattening C code

Flattening is a program transformation that, like inlining, removes edges from the callgraph by absorbing called functions into their callers' bodies. Unlike inlining, flattening does not create multiple copies of function bodies in the common case.

2.1 Flattening Algorithm

From the point of view of the callee (see Figure 1), flattening a single function works as follows:

1. Rename symbols in the callee to avoid name conflicts.

Before flattening:

```
int foo (int formal1, int formal2) {
  ... body ...
  return ret;
}
```

After flattening:

```
_foo_body:
  ... body ...
  _foo_ret_value = ret;
  switch (_foo_ret_site) {
    case 0:  goto _foo_ret_site_0;
    case 1:  goto _foo_ret_site_1;
    case 2:  goto _foo_ret_site_2;
    ...
    default: panic();
  }
```

Figure 1. Flattening from the callee's perspective

Before flattening:

```
val = foo (actual1, actual2);
```

After flattening:

```
_foo_formal1 = actual1;
_foo_formal2 = actual2;
_foo_ret_site = 2;
goto _foo_body;
_foo_ret_site_2:
  val = _foo_ret_value;
```

Figure 2. Flattening from the caller's perspective

2. For each local variable in the callee, move it to the caller.
3. Create a fresh variable for storing the return site.
4. Copy the callee's body into the caller, below a label.
5. Create a switch statement returning to each possible callsite. We include a default case in each such switch statement that crashes the running program—this can only happen in the presence of a compiler bug, flattener bug, or memory safety violation in the compiled application.

From the point of view of the caller (see Figure 2), flattening works as follows:

1. Create assignment statements connecting formal and actual parameters.
2. Create a unique numerical identifier for the callee's return point and an assignment statement storing the identifier for this callsite.
3. Create a jump to the callee's body.
4. Create a label for the return site.
5. Assign the function's return value into the appropriate variable.

Figure 3 shows a complete example: the body of A() is copied into B() and function calls are replaced with jumps.

Our flattening algorithm does not contain a special case for generating better code for functions that have a single callsite. This would be straightforward, but we solve the problem in a different way: by previously performing a function inlining pass that eliminates all functions that have a single callsite (see Section 2.5).

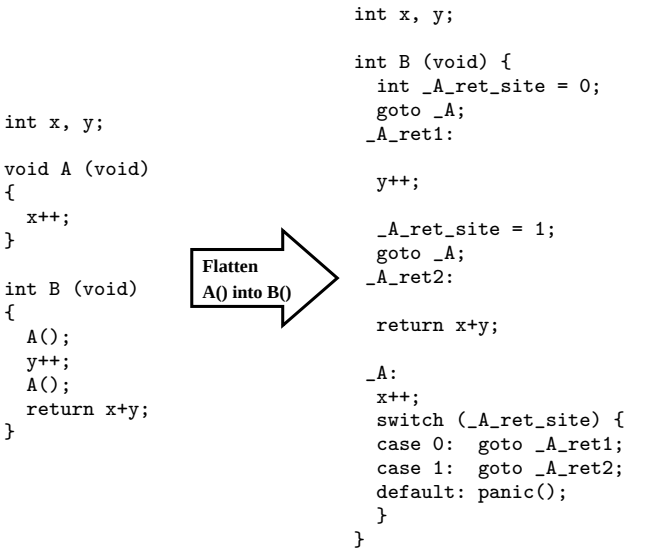

Figure 3. Example of flattening

2.2 Flattening Hazards

A *flattening hazard* is a condition that prevents flattening a particular callee into its caller. Hazards that we have identified are:

1. call through function pointer,
2. unavailable function body,
3. use of `setjmp` or `longjmp` in the callee,
4. explicit modification of stack pointer in the callee, for example to spawn a new thread or to dynamically allocate storage using `alloca`, and
5. recursive loops.

When a program contains multiple callgraphs, for example due to the presence of threads or interrupt handlers, each callgraph can be individually flattened. Flattening does not provide a way to merge these separate callgraphs.

When a program is free of flattening hazards, all function calls can be flattened. When a program additionally contains only a single callgraph, all variables can be lifted into `main`. At this point, the compiler is free to statically allocate the stack frame for `main`, eliminating all dynamic use of the stack. Furthermore, on a RISC architecture, the compiler is free to use the stack and frame pointers for other purposes (no compiler that we are aware of takes advantage of this opportunity, however).

2.3 Avoiding Code Duplication

Flattening a callee into a single calling function results in the creation of a single copy of the callee, regardless of the number of callsites in the caller. Flattening a callee into multiple callers causes one copy of the callee to be created for each caller. However, in the common case, code duplication can be avoided by (1) flattening only into the root function of each callgraph and (2) removing the original function once all calls have been flattened away. This leads to a useful result:

Theorem 1. *A hazard-free C program that contains a single callgraph (i.e., no interrupts or threads) can be completely flattened without introducing any duplicated code.*

Proof Sketch. Proof is by induction over the callgraph.

- Base case: `main` has no callees.
- Inductive case: `main` has a callee that we will flatten.
  - Subcase 1: A copy of this function has not yet been integrated into `main`. We flatten the function into `main`, creating one new copy of its body. However, this copy is "free" because we will eventually be able to eliminate the standalone version of the function body.
  - Subcase 2: A copy of the function has already been integrated into `main`. We add a new callsite to the function; no code is duplicated.

□

When functions are reachable from multiple callgraphs, a policy decision must be made about whether or not to flatten these functions. We do so in order to save as much RAM as possible—and pay for this with code size.

2.4 Lifting Global Variables

Once an embedded application has been flattened, some global variables can be "lifted" into the local scope of `main`. This transformation is sound when the global is referenced—directly and indirectly—only by `main`. Our flattener uses a simple alias analysis to verify this property for each global variable before lifting it into `main`'s scope. Additionally, since local variables in C are not automatically initialized, lifted variables lacking initializers must be explicitly initialized to zero. Globals are not lifted when they have external linkage or are specified (using a non-portable compiler directive) to reside in ROM instead of RAM.

2.5 Implementation

We implemented flattening and lifting as source-to-source transformations using CIL [Necula et al. 2002]. The algorithm is as follows:

1. Perform a function inlining pass to eliminate functions that are very small or that are called from just one site.
2. Flatten the program and lift global variables as described in this section.
3. Run a cleanup pass that performs dead code elimination, dead data elimination, and copy propagation.

The resulting C program can be passed directly to the embedded C compiler. Our flattener/lifter was implemented in 664 lines of OCaml code. In comparison, our source-to-source inliner is 1783 lines and our cleanup pass is 1011 lines.

3. Optimizing Flattened Code

The main purpose of flattening and lifting is to create additional opportunities for existing compiler optimizations to improve code. However, these opportunities also represent a hazard: it is possible that the compiler will not be up to the task of optimizing the much larger post-flattening code. Specifically, not only does flattened and lifted code generate considerable register pressure, but it also confuses a standard live range analysis by creating false paths that increase apparent register pressure. The compiler that we chose to improve is a pre-release snapshot of GCC 4.4.0 for the AVR architecture. We expect that these improvements would also be useful if implemented in other embedded compilers.

3.1 Problem 1: Flattening Creates False Paths

Flattening can create very large function bodies with large numbers of variables. For the resulting object code to be of high quality, it is critical that the compiler's register allocator do a good job. For the register allocator to do a good job, as few variables as possible

UU IR Author Manuscript

```
extern
volatile
int TIMER;

void C (void)
{
  ... no calls ...
}

void A (void)
{
  int a1 = TIMER;
  C();
  a2 = a1;
}

void B (void)
{
  int b1 = TIMER;
  C();
  b2 = b1;
}

void D (void)
{
 ...
 A();
 ...
 B();
 ...
}
```

```
extern
volatile
int TIMER;

void D (void) {
 int a1, b1;
 ...
_A_body:
 a1 = TIMER;
 _C_ret_site = 1;
 goto _C_body;
_c_ret_site_1:
 a2 = a1;
 ... A return code ...
_B_body:
 b1 = TIMER;
 _C_ret_site = 2;
 goto _C_body;
_c_ret_site_2:
 b2 = b1;
 ... B return code ...
_C_Body:
 ... C body ...
 switch (_C_ret_site) {
 case 1: goto _C_ret_site_1;
 case 2: goto _C_ret_site_2;
 ...
}
```

Figure 4. Example illustrating the introduction of false paths via flattening. At left, hypothetical source code including reads from a hardware register TIMER. At right, fragments of D() after A(), B(), and C() have all been flattened into it. In the original code it is obvious that `a1` and `b1` have non-overlapping live ranges. In the flattened code, many compilers will consider `a1` and `b1` to conflict.

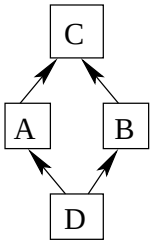

Figure 5. Callgraph for unflattened code in Figure 4

should *conflict*. Conflicts occur when the live ranges of two variables overlap at at least one program point. If two variables conflict, they may not be allocated to the same register or memory cell. Thus, to keep RAM usage low, it is critical for live ranges to be as short as possible.

Figure 4 shows how flattened code can trick a compiler into computing overly pessimistic live ranges. Given the callgraph for the original code (shown in Figure 5), it is obvious that `a1` and `b1` are not live at the same time. Flattening does not change the live range of any variable. However, a standard flow sensitive, path insensitive live variables analysis will be confused by the flattened code. The problem is that there are two paths through the code from C(): one returning to A()'s code, the other returning to B()'s code. Lacking an expensive path-sensitive live variables analysis, the analysis state is "polluted" by passing through C(), and it appears that `a1` and `b1` have overlapping live ranges.

We examined some embedded applications and found that when using an unmodified GCC to compile flattened code, 40%–70% of conflicts between variables were false positives.

3.2 Solution 1: CGO—Callgraph-Based Optimization

A variable in function $f()$ can conflict with a variable in function $g()$ only if there is a path through the callgraph leading from $f()$ to $g()$ or vice versa. Flattening a C program does not add any conflicts, but as we saw in Figure 4, flattening makes it difficult for the compiler's conflict analysis to succeed.

We solved this problem in a direct way: by reusing information about unflattened code to improve compilation of flattened code. Basically, by looking at unflattened code we can infer many non-conflict relationships between variables. We use this information to break apparent conflicts in the flattened code. In detail:

- We modified a GCC pass that performs call graph analysis to dump the callgraph of an unflattened application into a file, along with a mapping of variables to the functions in which each variable is declared.
- We modified GCC to read the callgraph and variable mapping information while compiling a flattened program. It computes a set of variable pairs that cannot be conflicts, and uses that information to break false conflicts.

3.3 Problem 2: Spilling Wastes RAM

Typically, a variable is put into memory, as opposed to being allocated to a register, if one of the following conditions is true:

- The variable has an aggregate type (struct, union, or array).
- The variable has its address taken and the resulting pointer is used in a nontrivial way.
- The variable spills out of the register allocator.

In general, on architectures with a reasonable number of registers, register allocator spills are uncommon. Thus—practically speaking—compilers do not seem very concerned with generating efficient code in the presence of spills. For example, it is common for spilled variables to effectively have infinite live ranges: the compiler makes no attempt to share their memory cells with other, non-conflicting spilled variables. Stack memory allocation is therefore very simple: variables are lined up according to their alignment requirements.

3.4 Solution 2: SSO—Stack Slot Optimization

Since flattened code is nearly certain to cause the register allocator to spill, it is important to avoid pathological behavior such as infinite live ranges for spilled variables. Quite recently (i.e., since GCC 4.3), GCC has started to optimize spilled registers by co-locating them on the stack if they fail to conflict [Makarov 2007].

We modified GCC to push this idea further by performing liveness analysis on variables with aggregate type and on variables manipulated using pointers, and then to collocate these variables on the stack when they fail to conflict. The analysis that we implemented is field sensitive. The goal is to minimize RAM consumption by co-locating non-conflicting variables whenever possible.

Our analysis finds, at the level of GCC's RTL (register transfer language), all references to stack memory locations. For each such reference, it computes a conservative set of possible target memory locations. Some cases that we deal with, where FP is the frame pointer, are:

- Addresses of the form FP $+ i$, where i is a constant. We can trivially resolve the location of the reference.
- Addresses of the form FP$+R_n$, where R_n is a register. Here we need to reason about the value of R_n; we do this by performing

a backwards dataflow analysis that, if successful, provides a concrete value for the register. For example consider this code:

```
if (R4)
   R2 = 0;
else
   R2 = 4;
if (R5)
   R1 = R2 * 2;
else
   R1 = R2 * 3;
load MEM(FP + R1)
```

The goal of our analysis is to find the value in R_1 when the load is performed. A backwards dataflow analysis can conclude that at the location of the memory reference, R_1 can contain 0, 8, or 12. In other words, the access is either 0, 8, or 12 bytes offset from the frame pointer. Our analysis includes special cases to deal with array induction variables; in this case the entire array being traversed is treated as being referenced by the eventual memory operation.

- Addresses of the form R_n. Again, we perform a backwards dataflow analysis, with the additional complication that the target may not be on the stack. If the analysis terminates without involving the frame pointer, then the access is to a global variable and we may disregard it. If the access may be to the stack, analysis proceeds as for the second case.

In some cases, source-level symbolic information can be used to help our analysis. For example, for an RTL instruction `store [Rn]` where R_n is a pointer, and $[R_n]$ is associated with the location `X[3].f`, we can simply mark the corresponding struct field as being referenced. If the array index is indeterminate, the entire array would be marked as being referenced. Finally, if our various heuristics and analysis fail to compute a useful, conservative set of memory locations that may be referenced by a load or store, we treat it as possibly referencing all memory locations that are part of objects whose address has been taken.

Unlike GCC's existing liveness analyses, our new conflict analysis deals with may-alias relations. Thus, in addition to def/use information, we have may-def and may-use information. We deal with these in the standard way: a must-define command marks the beginning of a live range, but a may-define does not. A may-use command can be treated the same as a must-use command.

Once our analysis has computed live ranges for stack slots, it performs a conflict analysis. A spilled register may share memory locations with any kind of stack object with which it fails to conflict, including unused fields of live structs.

4. Evaluation

This section evaluates flattening, lifting, and our compiler improvements using a collection of embedded applications written in C and targeting the AVR architecture. The AVR is an 8-bit RISC architecture; like other ultra-low-power microcontrollers, it lacks performance-oriented features such as caches and branch predictors.

4.1 Test Applications

Our test applications are:

- FlyByWire—an unmanned aerial vehicle control program from the Paparazzi project [Paparazzi]. It is responsible for reading and decoding sensor data, driving servos, and switching between manual and automatic control. It is 1306 lines of code (LOC).

application	baseline RAM	baseline ROM
FlyByWire	1174	9516
Robot	290	3386
Projector	78	9438
BaseStation	1747	12156
Blink	111	1968
BlinkToRadio	477	9366
Oscilloscope	512	9846
RadioCountToLeds	483	9362
SharedResourceDemo	136	2616
TestAM	449	8878
TestFcfsArbiter	126	2306
TestLocalTime	405	5580
TestRoundRobinArbiter	128	2414
TestSerial	407	5652
TestSimComm	475	9180

Figure 6. Baseline RAM and ROM usage in bytes

- Robot—a robot control application emitted by KESO [Wawersich et al. 2007], an ahead-of-time Java-to-C compiler for highly constrained embedded systems. 2526 LOC.
- Projector—a laser video projector control program [Hjortland]. 2307 LOC.
- 12 sensor networking applications from TinyOS [TinyOS] 2.1, emitted by the nesC [Gay et al. 2003] compiler. 3015–13978 LOC.

Although these applications target different members of the AVR processor family, the TinyOS applications all target the ATmega128, a chip supporting 4 KB of SRAM and 128 KB of flash memory. Figure 6 shows the default RAM and ROM usage of these applications.

4.2 Evaluation Metrics

Our primary evaluation metrics are ROM usage, RAM usage, and CPU usage. ROM usage is trivial to measure since the size of an MCU application's code segment is static.

Estimating RAM Usage A C program's static RAM is divided into two segments. The *data segment* holds global variables that are explicitly initialized. The *BSS segment* holds global variables that lack initializers; this segment is zeroed (as required by the C standard) at boot time. The sizes of both are fixed at link time and are therefore trivial to compute.

Estimating the worst-case stack memory usage of a program is not totally straightforward, but the problem has been previously addressed [Brylow et al. 2001, McCartney and Sridhar 2006, Regehr et al. 2003]. Our approach to bounding stack memory consumption is based on this previous work: we have a program that walks the callgraphs of an AVR binary, conservatively estimating the worst-case stack depth of each function as it is found. To compute total worst-case stack depth, the worst-case depth of `main` is added to the maximum stack depth of any interrupt handler.

For sequential code, such as `main` or an isolated interrupt handler, our stack analyzer tends to return tight bounds: its estimates are in close agreement with actual measurements. For concurrent code, such as an entire sensor network application, our stack analyzer typically returns stack memory bounds that are larger (often by a factor of two) than the worst-observed stack memory usage. We believe that our analyzer is correct. The gap between predicted and observed maximum stack memory usage is due to the fact that reaching a concurrent system's maximum stack depth requires timing coincidences that have low probability in practice.

UU IR Author Manuscript

The RAM usage of an application is the sum of the RAM usage of the data segment, the BSS segment, and the call stack. None of our test applications uses a heap, and in fact heaps are uncommon on microcontrollers with just a few KB of RAM.

CPU Usage The standard metric for a sensor network application's CPU usage is *duty cycle*: the fraction of time that the processor is active, as opposed to sleeping in a power-saving mode. We measured duty cycle using Avrora [Titzer et al. 2005], a cycle-accurate simulator for sensor network applications.

We measured the CPU usage of only the sensor networking applications, omitting Robot, Projector, or FlyByWire from our duty cycle evaluation. We lack the customized hardware—and simulators for that hardware—required by these embedded applications.

4.3 Methodology

The baseline for comparison is an unimproved version of GCC 4.4.0 for the AVR architecture. The following factors help ensure a fair comparison between GCC and our flattener:

- The nesC compiler emits a single C file for an application and also aggressively marks functions with the "inline" directive, which GCC obeys.
- For non-TinyOS applications, we used CIL to combine all source files pertaining to each application into a single C file.

These measures help the unmodified GCC by permitting it to see the entire application at once, and for example to inline arbitrary calls even when these cross compilation units in the original application. These measures do not help the flattener, which performs file combining on its own.

We evaluated seven compilations of each application:

1. Baseline: unmodified source code, compiled with unmodified GCC 4.4.0
2. Flattened: flattened source code, compiled with unmodified GCC 4.4.0
3. Flattened + SSO: flattened source code, compiled with GCC 4.4.0 and stack slot optimizations (Section 3.4)
4. Flattened + SSO + CGO: flattened source code, compiled with GCC 4.4.0, stack slot optimizations, and callgraph-based optimization (Section 3.2)
5. Unflattened + SSO
6. Unflattened + CGO
7. Unflattened + SSO + CGO

The last three combinations of optimizations resulted in identical resource usage compared to the baseline, and so we omit these data points from our graphs. SSO (stack slot optimization) is ineffective when compiling unflattened applications, which cause few or no register allocator spills, and CGO (callgraph optimization) clearly has no effect since the compiler already has all of the callgraph information for the unflattened code. Our results are shown in Figure 7. Each graph contains three data points for each application: change due to flattening vs. the baseline, change due to flattening + SSO vs. the baseline, and change due to flattening + SSO + CGO vs. the baseline.

4.4 Impact on RAM Usage

On average, the RAM usage of our test applications was reduced by 20% when compiled with flattening, CGO, and SSO. This is our most significant high-level result: RAM is in very short supply on a low-power MCU.

Flattening alone reduces RAM usage by 15%, on average. In one case, flattening increased RAM usage slightly due to greatly increasing register pressure, causing the compiler to spill data to RAM. However, in this case SSO and CGO were effective in bringing RAM usage down below the baseline.

application	variables lifted	bytes lifted
FlyByWire	72%	45%
Robot	75%	97%
Projector	0%	0%
BaseStation	20%	80%
Blink	44%	78%
BlinkToRadio	22%	54%
Oscilloscope	24%	52%
RadioCountToLeds	22%	46%
SharedResourceDemo	72%	74%
TestAM	18%	49%
TestFcfsArbiter	67%	80%
TestLocalTime	14%	30%
TestRoundRobinArbiter	64%	79%
TestSerial	14%	28%
TestSimComm	21%	46%

Figure 8. Success rates for lifting global variables into `main`'s scope after flattening

4.5 Impact on Duty Cycle

On average, the CPU usage, measured using duty cycle, is reduced by 3% by flattening. However, flattening increases the CPU usage of several applications. SSO and CGO have little or no effect on CPU usage.

Flattening can speed up a program by effectively permitting the compiler to use customized calling conventions and by increasing the scope of existing intraprocedural optimizations. On the other hand, returning from flattened function calls is not as efficient as returning from real function calls. Furthermore, the increased function size of a flattened program causes the compiler to use more long branches, which are inefficient.

4.6 Impact on ROM Usage

The biggest drawback of flattening is that it increases code size, by an average of 14% when flattening, CGO, and SSO are all applied to a program. There are several contributors. First—and most important—return sites from functions with many callsites can become quite bulky. For example, we saw functions in large TinyOS applications with up to 27 call sites. Second, flattening requires code to be inserted to explicitly initialize global variables that are lifted into `main`. Third, when a function is flattened both into `main` and into an interrupt handler, its code is duplicated.

Despite the code bloat, we believe that flattening is useful for RAM-constrained applications, particularly on architectures such as AVR that have many times more ROM than RAM.

4.7 Lifting Globals

Figure 8 shows the percentage of variables and bytes that could be lifted into `main`'s local scope after each application was flattened. Lifting is important because it exposes variables to more aggressive optimizations. Across the applications that we tested, the main reason why a variable could not be lifted into `main`'s scope was that it was potentially accessed from an interrupt handler. For example, the Projector application contains 16 global variables, all of which are shared with interrupts.

4.8 Compile Times

The optimizations presented in this paper—flattening, SSO, and CGO—are all fairly fast. The flattener, which includes function

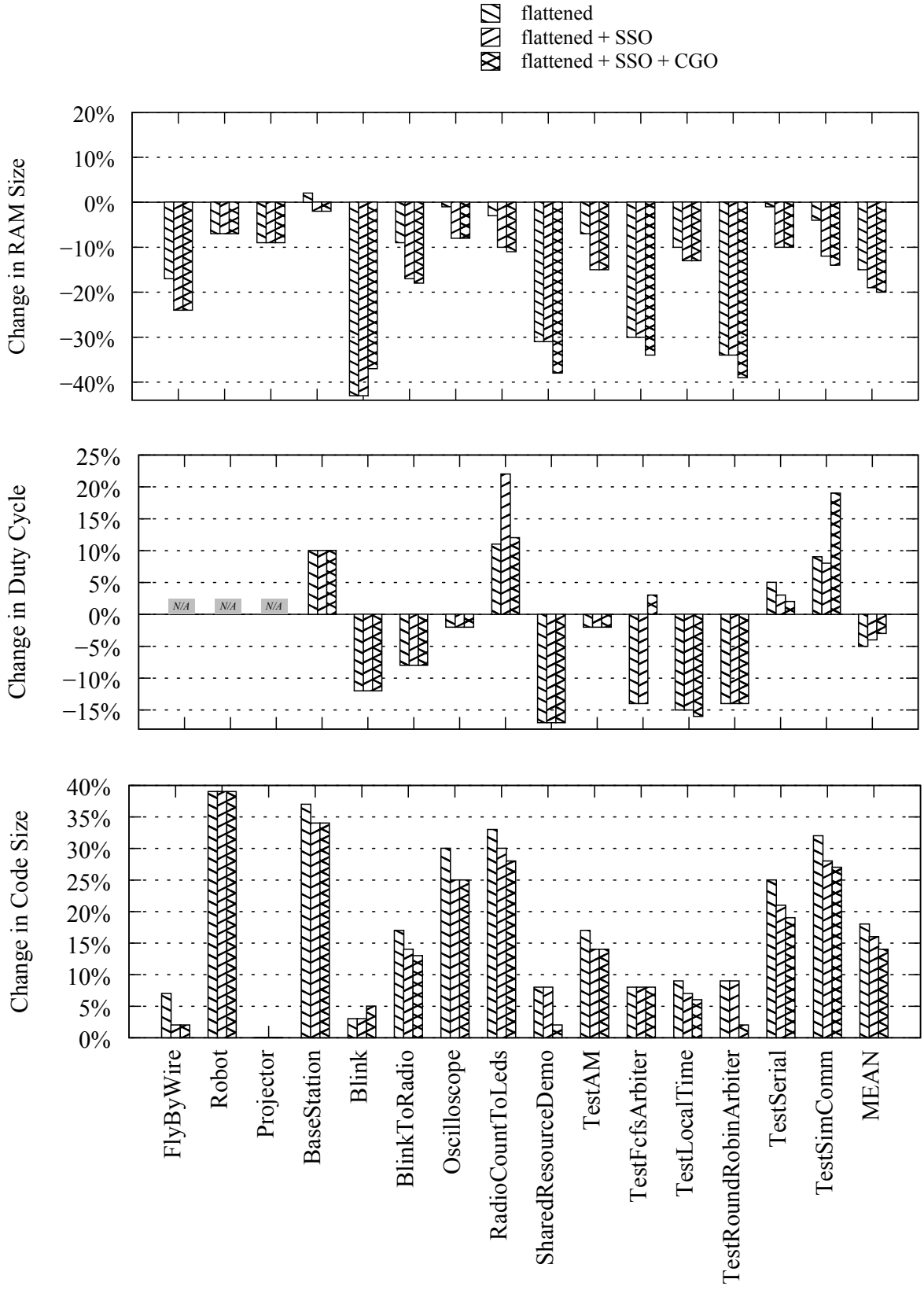

Figure 7. Effect of flattening on resource usage for our test applications. The baseline for comparison is compilation with inlining targeted towards minimizing code size.

inlining and dead code/data elimination passes, takes up to one minute to run on our largest testcases. However, this is an entirely unoptimized prototype implementation written in a functional language—we believe that flattening could be made to be very fast, adding perhaps a few percent to overall compile times. SSO and CGO add no more than about three seconds to the compile time of our largest test applications.

5. Discussion

This section addresses some of the broader implications of our research.

5.1 Toward Bounded Memory Usage

When a C program is completely flattened, a single stack frame, for `main`, results. Furthermore, the compiler is free to allocate this frame statically, and to never deallocate it. In fact, this is what some versions of GCC for the AVR architecture do.

In the absence of dynamic use of the callstack, the worst-case memory usage of an embedded application becomes easy to predict: it is the same as the best-case memory usage. This permits developers to avoid a difficult problem: predicting worst-case stack memory usage. Another way to avoid this problem is to use a static analysis tool [Brylow et al. 2001, McCartney and Sridhar 2006, Regehr et al. 2003]. However, for various reasons these tools are not in widespread use and in practice developers rely on guesswork [Ganssle 1999].

Even when complete flattening is impossible—as it often is, due to interrupt handlers and library calls—flattening greatly simplifies an application's callgraph. Thus, while flattening does not eliminate the stack depth estimation problem, it does make it easier to compute a program's worst-case stack memory usage.

5.2 Control Flow Integrity

Control flow integrity (CFI) [Abadi et al. 2005] is a program property specifying that execution must follow a control flow graph determined ahead of time. CFI means that a program is invulnerable to *code injection attacks* where an attacker exploits a buffer overflow vulnerability to jump to malicious code residing in a data buffer. Recent work [Francillon and Castelluccia 2008] has shown that even Harvard architecture MCUs, which store code in ROM, may be vulnerable to code injection attacks.

For programs in unsafe languages like C and C++, static verification of CFI is difficult or impossible due to problems in reasoning about pointers and arrays. Enforcement of CFI, then, must be dynamic. A major problem is that regular program data is mixed up on the stack with metadata such as saved return addresses. A typical enforcement mechanism for CFI involves creating a *shadow stack* which stores only metadata, and protecting it from being overwritten using sandboxing techniques.

Unlike a regular C program, a completely flattened C program does not store function return addresses anywhere in memory. Instead, small integers denoting return sites are stored in regular program variables, which may or may not be spilled to RAM. If a function return site is corrupted by a buffer overflow vulnerability while it resides in RAM, CFI is not violated. Rather, as the code in Figure 1 shows, the worst thing an attacker can accomplish is to make the program panic or return to the wrong caller. Although these consequences are undesirable, they are certainly safer than executing arbitrary injected code.

A bit more formally:

Theorem 2. *A hazard-free C program that contains only vanilla C control flow constructs (i.e., no `setjmp`/`longjmp`, UNIX signals, interrupts, coroutines, etc.) has the control flow integrity property when completely flattened and compiled by a correct C compiler.*

Proof Sketch. Assume the CFG is a set of addresses that the program counter (PC) may legally execute, and also that the executing program cannot overwrite any instruction in the CFG. Proof is by induction over the PC successor relation.

- Base case: PC is at the first instruction of the program, which is trivially in the CFG.
- Inductive case: If the PC is in the CFG, control flow will transfer to another instruction in the CFG.
  - Subcase: Next instruction is reached via standard fallthrough. Successor is trivially in the CFG.
  - Subcase: Next instruction is reached via a direct conditional or unconditional branch. Again, the successor instruction(s) must be in the CFG. If the program has managed to corrupt its state via buffer overflow we may jump to the wrong target, but CFI is not violated.
  - Subcase: Next instruction is reached via indirect branch. Here we must look at what C code caused the indirect branch to be generated. By assumption, the program contains no function pointers or exotic control flow constructs. Given vanilla C code, most compilers will emit an indirect branch only to implement a sufficiently large `switch` statement. As far as we know, all compilers' jump table implementations respect CFI even after a program's state has been corrupted by a buffer overrun.
  - Subcase: Next instruction is reached via function call, function return, interrupt, or interrupt return. By assumption, none of these instructions are in the CFG.

□

Although our implementation of CFI handles only a subset of C programs, it results in more efficient CFI enforcement than any other CFI technique of which we are aware.

5.3 Future Work

Flattening interrupt handlers Our current implementation flattens `main` and any interrupt handlers separately. As we noted in Section 4.7, this prevents us from lifting globals into `main` when they are reachable from interrupts. An alternative that we plan to explore is flattening interrupt-handling code into `main`. Note, however, that this scheme does not let us avoid code duplication for functions that are called from `main` and also from an interrupt handler: this duplication is required in order to ensure that fresh versions of local variables are available for interrupt handlers. Flattening interrupt handlers into `main` will not work for embedded applications with *reentrant* interrupts, which may have multiple concurrent activations.

Total stack elimination Although we can eliminate all dynamic use of the stack for hazard-free C programs, a legacy C compiler will still devote a register to the stack pointer and (usually) another to the frame pointer. We would like to modify a compiler for a RISC architecture to recover these registers for general-purpose use in the case where a program can be completely flattened.

Selective flattening As we have shown, sometimes over-aggressive flattening can result in too many register spills, wasting RAM. We plan to explore fine-tuning our flattener by making it sensitive to the number of conflicting temporaries it generates. We expect that there are interesting heuristic choices to make when parameterizing a selective flattener: it should be sensitive to the shape of the callgraph, the number of registers in the target architecture, etc.

6. Related Work

An enormous body of research exists on manipulating program control flow and optimizing memory layout. Here we compare our work to some representative examples.

Through Fortran 77, Fortran did not support recursive function calls: programs could be compiled into stackless object code. More recent versions of Fortran have supported a stack. In relation to compilation techniques for functional languages, flattening could be viewed as an aggressively optimized, local, environmentless continuation passing style (CPS) [Appel 1992] conversion for non-recursive programs. Like function inlining [Cooper et al. 1991] and cloning [Cooper et al. 1993], flattening copies code and alters control flow in simple ways, with the goal of enabling subsequent optimizations to do a better job.

Biswas et al. [2004] and Middha et al. [2005] use compiler-driven techniques to blur the lines between different storage regions. This permits, for example, stacks to overflow into unused parts of the heap, globals, or other stacks. CRAMES [Yang et al. 2006] saves RAM by applying standard data compression techniques to swapped-out virtual pages, based on the idea that these pages are likely to remain unused for some time. MEMMU [Bai et al. 2006] provides on-line compression for systems without a memory management unit, such as wireless sensor network nodes. Ozturk et al. [2005] compress data buffers in embedded applications. Cooprider and Regehr [2007] performed compile-time RAM compression for TinyOS applications. We believe our work is largely orthogonal to these previous techniques, which exploit properties of an application's data to save memory. In contrast, flattening exploits the structure of a computation to improve memory layout.

A significant body of literature exists on changing the layout of objects in memory to improve performance, usually by improving spatial locality to reduce cache and TLB misses. Good examples include Chilimbi et al.'s work on cache-conscious structure layout [Chilimbi et al. 1999] and Rabbah and Palem's work on data remapping [Rabbah and Palem 2003]. In contrast, in our work locality is irrelevant: the MCUs that we target have flat RAM with a uniform access time.

Ananian and Rinard [2003] perform static bitwidth analysis and field packing for Java objects, with the goal of reducing memory usage. Zhang and Gupta [2006] use memory profile data to find limited-bitwidth heap data that can be packed into less space. Lattner and Adve [2005] save RAM through a transformation that makes it possible to use 32-bit pointers on a per-data-structure basis, on architectures with 64-bit native pointers. Chanet et al. [2005] apply whole-program optimization to the Linux kernel at link time, reducing both RAM and ROM usage. Virgil [Titzer 2006] has a number of RAM optimizations including reachable members analysis, reference compression, and moving constant data into ROM.

Barthelmann [2002] describes inter-task register allocation, a global optimization that saves RAM used to store thread contexts. Grunwald and Neves [1996] save RAM by allocating stack frames on the heap, on demand, using whole-program optimization to reduce the number of stack checks and to make context switches faster.

7. Conclusions

The high-level result of our work is that relatively simple compilation techniques can reduce the RAM usage of medium-sized microcontroller applications (1300–14,000 lines of code) by an average of 20%. This result is important because ultra-low-power microcontrollers have limited on-chip RAM and many applications are RAM-bound. On average, flattened applications use 3% less CPU time. The main cost of our transformation is code size: ROM usage is increased by an average of 14%. We believe that in many cases, this RAM/ROM tradeoff is a desirable one. For example, a typical member of the AVR family of microcontrollers that we use in our experimental evaluation has 32 times more ROM than RAM.

Our first contribution is the *flattening* program transformation which turns function calls into jumps, bypassing the calling convention and increasing the effectiveness of intraprocedural optimizations. Applied aggressively, flattening enables *global variable lifting* where variables with global scope can be moved into `main`'s local scope. When a program is free of flattening hazards, it can be completely flattened, resulting in *stack elimination* since the only remaining stack frame belongs to `main`, and it can be statically allocated.

Our second contribution is to show that flattening exposes weaknesses in existing compiler optimizations, and to fix some of these weaknesses. One problem is that flattened versions of functions with multiple callsites confuse typical path-insensitive live range analyses by adding false paths to a program. We developed a *callgraph-based optimization* that takes live range information from the original, unflattened program and uses it to break conflicts in the flattened program. Another problem is that flattening causes large increases in register pressure, resulting in spilling. Most compilers deal with spilled variables in a suboptimal way, treating them as being live for the duration of the enclosing function. We developed a *stack slot optimization* to enable collocation of non-conflicting stack variables.

In summary, we have shown that flattening can save significant amounts of RAM for embedded applications. However, to get the most benefit, coexisting compiler optimization passes must be specifically strengthened to cope with flattening-induced optimization challenges.

Acknowledgments

The authors would like to thank Yang Chen, Mary Hall, Matthew Might, Jon Rafkind, and Alastair Reid for their invaluable help and advice. This material is based upon work supported by the National Science Foundation under Grant Nos. 0615367 and 0448047, and by DARPA.

References

Martn Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control flow integrity: Principles, implementations, and applications. In *Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proc. of the 2003 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 59–68, San Diego, CA, June 2003.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, MA, 1992.

Lan S. Bai, Lei Yang, and Robert P. Dick. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Seoul, Korea, October 2006.

Volker Barthelmann. Inter-task register-allocation for static operating systems. In *Proc. of the Joint Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES) and Software and Compilers for Embedded Systems (SCOPES)*, pages 149–154, Berlin, Germany, June 2002.

Surupa Biswas, Matthew Simpson, and Rajeev Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Washington, DC, September 2004.

Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.

Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the Linux kernel. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proc. of the ACM SIGPLAN 1999 Conf. on Programming Language Design and Implementation (PLDI)*, pages 13–24, Atlanta, GA, May 1999.

Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, April 1993.

Nathan Cooprider and John Regehr. Offline compression for on-chip RAM. In *Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI)*, pages 363–372, San Diego, CA, June 2007.

Jakob Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symp. (RTAS)*, Vancouver, Canada, June 1999.

Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In *Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS)*, Alexandria, VA, October 2008.

Jack Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1999.

David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 50–59, Cambridge, MA, October 1996.

Håkon A. Hjortland. Laser video projector. `http://heim.ifi.uio.no/haakoh/avr/`.

Chris Lattner and Vikram Adve. Transparent pointer compression for linked data structures. In *Proc. of the ACM Workshop on Memory System Performance (MSP)*, Chicago, IL, June 2005.

Vladimir N. Makarov. The integrated register allocator for GCC. In *Proc. of the GCC Developers' Summit*, pages 77–90, Ottawa, Canada, July 2007. URL `http://ols.fedoraproject.org/GCC/Reprints-2007/makarov-reprint.pdf`.

William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2006)*, pages 167–180, Boulder, Colorado, October 2006.

Bhuvan Middha, Matthew Simpson, and Rajeev Barua. MTSS: Multi task stack sharing for embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, September 2005.

George C. Necula, Scott McPeak, S. P. Rahul, and Westley 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, April 2002.

Ozcan Ozturk, Mahmut Kandemir, and Mary Jane Irwin. Increasing on-chip memory space utilization for embedded chip multiprocessors through data compression. In *Proc. of the 3rd Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 87–92, Jersey City, NJ, September 2005.

Paparazzi. The Paparazzi project, 2006. `http://www.nongnu.org/paparazzi`.

Rodric M. Rabbah and Krishna V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Trans. Embedded Computing Systems (TECS)*, 2(2):1–32, May 2003.

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.

TinyOS. tinyos.net. `http://www.tinyos.net`.

Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 191–208, Portland, OR, October 2006.

Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, pages 44–53, Los Angeles, CA, April 2005.

Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. An OSEK/VDX-based Multi-JVM for automotive appliances. In *Proc. of the Intl. Embedded Systems Symposium*, Irvine, CA, 2007.

Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. On-line memory compression for embedded systems. *ACM Trans. Embedded Computing Systems (TECS)*, 2006.

Youtao Zhang and Rajiv Gupta. Compressing heap data for improved memory performance. *Software—Practice and Experience*, 36(10):1081–1111, August 2006.