

Offline Compression for On-Chip RAM

Nathan Coopriider John Regehr

School of Computing, University of Utah

{coop,regehr}@cs.utah.edu

Abstract

We present *offline RAM compression*, an automated source-to-source transformation that reduces a program's data size. Statically allocated scalars, pointers, structures, and arrays are encoded and packed based on the results of a whole-program analysis in the value set and pointer set domains. We target embedded software written in C that relies heavily on static memory allocation and runs on Harvard-architecture microcontrollers supporting just a few KB of on-chip RAM. On a collection of embedded applications for AVR microcontrollers, our transformation reduces RAM usage by an average of 12%, in addition to a 10% reduction through a dead-data elimination pass that is also driven by our whole-program analysis, for a total RAM savings of 22%. We also developed a technique for giving developers access to a flexible spectrum of tradeoffs between RAM consumption, ROM consumption, and CPU efficiency. This technique is based on a model for estimating the cost/benefit ratio of compressing each variable and then selectively compressing only those variables that present a good value proposition in terms of the desired tradeoffs.

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 data compression, embedded software, memory optimization, static analysis, TinyOS, sensor networks

1. Introduction

In 2004, 6.8 billion microcontroller units (MCUs) were shipped [5]: more than one per person on Earth. MCUs are small systems-on-chip that permit software control to be added to embedded devices at very low cost: a few dollars or less. Sophisticated electronic systems, such as those running a modern automobile, rely heavily on MCUs. For example, in 2002 a high-end car contained more than 100 processors [4].

RAM constraints are a first-order concern for developers. A typical MCU has 0.01–100 KB of RAM, 4–32 times as much ROM as RAM, and no floating point unit, memory management unit, or caches. We refer to program memory as “ROM” since it is treated as such by applications, even though it can usually be written to (but slowly and in blocks, in the common case of flash memory). Pro-

grams running on microcontrollers are usually limited to on-chip RAM. The cost of adding off-chip RAM is high, and many MCUs do not even provide an external memory bus. RAM constraints are one of the main reasons why MCU software does not commonly take advantage of useful abstractions such as threads and a heap.

We developed *offline RAM compression*, a source-to-source transformation that reduces the amount of memory used by a C program by encoding and bit-level packing global scalars, pointers, arrays, and structures based on the results of static whole-program analysis. The goal of our work is to reduce the RAM usage of existing embedded software without a great deal of overhead and in a way that is predictable at compile time. Our work targets legacy C code and reduces RAM requirements automatically and transparently. Compile-time predictability rules out *online RAM compression*, a family of techniques that find and exploit redundant data as a system executes [7, 32].

1.1 Fundamental observations

The following properties of embedded systems motivate our work.

RAM is used inefficiently. Although an n -bit machine word can store 2^n distinct values, in practice a typical word allocated by a program stores many fewer values. For example, Brooks and Martonosi [8] found that “roughly 50% of the instructions [in SPECint95] had both operands less than or equal to 16 bits.” To verify that small embedded systems behave similarly, we instrumented a simulator for Mica2 sensor network nodes to keep track of the number of distinct values stored in each byte of RAM. Bytes are a reasonable unit of measurement since Mica2s are based on the 8-bit AVR architecture. We then ran a collection of sensor network applications in the simulator. We found that, on average, a byte of RAM used to store a global variable (or part of one) took on just under four values over the execution of an application. In other words, six out of every eight bits of RAM allocated to global variables are effectively unused.

On-chip RAM is persistently scarce in low-end systems. RAM is, and always has been, in short supply for small embedded systems that are constrained by power, size, and unit cost. Furthermore, it is not at all clear that RAM constraints are going to go away in the foreseeable future. As transistor cost decreases, it becomes possible not only to create more capable processors at a given price point, but also to create cheaper, lower-power processors with a given amount of RAM. Since many sectors of the embedded market are extremely sensitive to unit cost, developers must often choose the smallest, cheapest, and lowest-power MCU that meets their needs. The decreasing cost of MCUs with a given set of capabilities supports the development of new applications, such as sensor networks, that were not previously feasible.

Value functions for resource use are discontinuous. For desktop and workstation applications, value functions for resource use are generally continuous in the sense that a small increase in resource

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006...\$5.00

Device	ROM	RAM	Ratio	Price
ATtiny24	2 KB	128 B	16:1	\$0.70
ATtiny45	4 KB	256 B	16:1	\$0.97
ATmega48	4 KB	512 B	8:1	\$1.50
ATmega8	8 KB	1024 B	8:1	\$2.06
ATmega32	32 KB	2048 B	8:1	\$4.75
ATmega128	128 KB	4096 B	32:1	\$8.75
ATmega256	256 KB	8192 B	32:1	\$10.66

Figure 1. Characteristics and prices of some members of Atmel’s AVR family, a popular line of 8-bit MCUs. Prices are from <http://digikeey.com> and other sources, for quantities of 100 or more.

use maps to a small decrease in utility. In contrast, given an embedded processor with fixed amounts of on-chip RAM and ROM:

- A program requiring too much of either kind of storage simply cannot be run: it has no value.
- Once a system fits into RAM and ROM, further optimization provides no additional value.

These discontinuities imply that it is important to be able to trade off between resources like RAM size, ROM size, and CPU use. The ratio of ROM to RAM on the MCUs in Figure 1 gives us a general idea of how much ROM we should be willing to sacrifice to save a byte of RAM. Of course, the actual “exchange rate” for a given system depends on what its limiting resource is.

Manual RAM optimization is difficult. Most embedded software developers—including ourselves—have had the experience of running out of RAM with features still left to implement. Manually reducing RAM usage is difficult and error-prone. Furthermore, it is not forward-thinking: highly RAM-optimized software often becomes over-specialized and difficult to maintain or reuse.

1.2 Benefits

From the point of view of an embedded software developer, our work has two concrete benefits. First, it can support basing a product on a less expensive, more energy-efficient MCU with less RAM. For example, the Robot2 application that is part of our benchmark suite (see Section 6) requires 368 bytes of RAM and runs on an Atmel ATmega8535 MCU with 512 bytes of RAM. The compressed version of Robot2 uses 209 bytes and therefore would fit onto a cheaper part with 256 bytes of RAM. Figure 1 gives an idea of the potential cost savings. Savings of pennies or dollars can add up to real money in the high-volume, low-margin markets for which MCUs are designed.

The second important benefit of our work is that it can enable substantially more functionality to be placed on an MCU with a given amount of RAM. For example, consider a sensor network developer who wishes to add network reprogramming [16] and link-level packet encryption [17] to an existing TinyOS application that already uses the entire 4 KB of RAM on a Mica2 sensor network node. Since these features are orthogonal to application logic, both are designed as transparent add-ons. Obviously this only works if enough memory is available. Across a set of representative TinyOS applications, our tool reduces RAM usage by an average of 19%—more than enough to support the addition of network reprogramming, which requires 84 bytes of RAM (2.1% of the total), and encryption, which requires 256 bytes of RAM (6.3% of the total). Without our work, this developer would be forced to manually reduce RAM usage by a total of 340 bytes: an unpleasant proposition at best.

1.3 Contributions

Our research has three main contributions:

1. Offline RAM compression, a new technique for automatically reducing RAM usage of embedded software using whole-program analysis and source-to-source translation.
2. A novel technique supporting flexible tradeoffs between RAM size, ROM size, and CPU efficiency by estimating the cost/benefit ratio of compressing each variable and then selectively compressing only the most profitable, up to a user-configurable threshold.
3. A tool, CComp, that implements offline RAM compression and tradeoff-aware compilation. Although CComp currently targets C code for AVR processors, only a small part of our tool is architecture-specific. CComp is available at <http://www.cs.utah.edu/~coop/research/ccomp/> as open-source software.

2. Background: Microcontroller-Based Embedded Software

Software for MCUs is somewhat different from general-purpose application code. These characteristics are largely programming-language independent, having more to do with requirements of the domain and properties of the platform. Below are some key features of microcontroller software; two represent complications that we are forced to handle and one is a restriction that we can exploit.

Software is interrupt-driven. Interrupts are the only form of concurrency on many MCU-based systems, where they serve as an efficient alternative to threads. Interrupt-driven software uses diverse synchronization idioms, some based on disabling interrupts and others based on volatile variables and developer knowledge of hardware-atomic memory operations. The implication for our work is that we need to provide a sound dataflow analysis of global variables even in the presence of unstructured, user-defined synchronization primitives.

Locality is irrelevant. MCUs have relatively fast memory, no caches or TLBs, and minimal (three stage or smaller) pipelines. Thus, there are no performance gains to be had by improving spatial locality through RAM compression and, furthermore, it is difficult to hide the overhead of compressing and uncompressing data. The implication is that code implementing compression must be carefully tuned and that we must invest effort in optimizations.

Memory allocation is static. The only form of dynamic memory allocation on most MCU-based systems is the call stack. When RAM is very constrained, heaps are unacceptably unpredictable (e.g., fragmentation is difficult to reason about and allocations may fail). The implication is that our pointer analysis can be in terms of static memory objects and stack objects.

3. Static Analysis

CComp is our tool that performs offline RAM compression through static analysis and source-to-source transformation. It borrows heavily from cXprop, our existing whole-program dataflow analyzer for C [12]. cXprop is itself built upon CIL [20], a parser, typechecker, and intermediate representation for C. Figure 2 shows the role of CComp in the toolchain.

CComp is sound under a standard set of assumptions. Mainly, the analyzed program must not perform out-of-bounds memory accesses, and inline assembly must be well-behaved (no side effects visible at the source level).

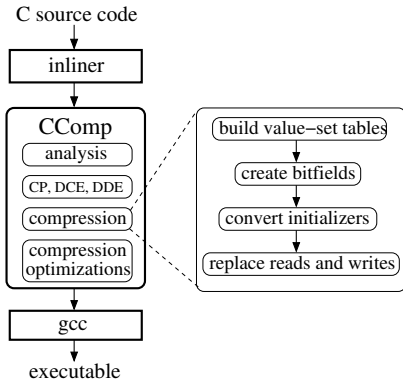


Figure 2. Toolchain for offline RAM compression

This section describes both the previous analysis features that we borrowed from cXprop and the new ones implemented by CComp.

3.1 Inlining

Before CComp proper runs, we run the code being analyzed through an aggressive source-to-source function inlining pass. Our inliner is based on a model of code bloat; its goal is to minimize the size of the eventual object code. Inlining increases the size of functions, which improves the precision of our context-insensitive dataflow analysis.

3.2 Value set analysis

Offline RAM compression is driven by the *value set* abstract domain, where abstract values are bounded-size sets of concrete values. From cXprop, CComp inherited transfer functions not only for the value set domain but also for the interval and bitwise domains. The interval domain represents abstract values as integer ranges, for example $[16..200]$. In the bitwise domain, each bit of a value is independently \perp (unknown), 0, or 1. Our choice of the value set domain was pragmatic: of the domains we implemented, it produces the best results. Interval-domain analysis of our benchmark programs supports 1.8% RAM compression and bitwise analysis supports 2.2% RAM compression, whereas value set analysis supports 12% RAM compression. Note, however, that our implementations of these three domains are not directly comparable: we have extensively tuned our value set implementation for precision, for example by adding *backwards* transfer functions that gain information from branches.

The maximum value set size is configurable; for this work, we set it to 16. Larger sizes resulted in significant analysis slowdown and only a minor increase in precision.

3.3 Pointer set analysis

The points-to analysis in CComp is based on a *pointer set analysis* that maintains explicit sets of locations that each pointer may alias. This analysis is fairly precise because it is flow-sensitive and field-sensitive. The pointer set domain works well for MCU-based applications because the set of items that are potentially pointed to is generally fairly small. A pointer set of cardinality one indicates a must-alias relationship, supporting strong updates of pointed-to values (a strong update replaces the existing state at a storage location, whereas a weak update merges the new information with the old). The pointer set domain, illustrated in Figure 3, is analogous to the value set domain except that it contains a special *not-NULL* element that permits the domain to represent the (common) case where nothing is known about a pointer’s value except that it cannot be NULL.

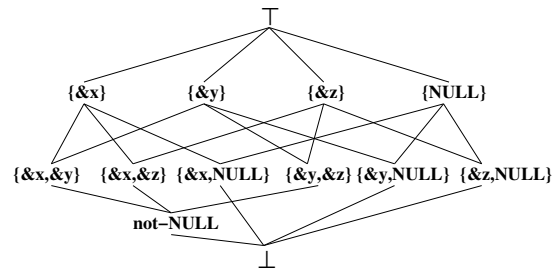


Figure 3. Pointer set lattice for three variables and maximum set size of two

3.4 Concurrency analysis

CComp can soundly analyze global variables even in the presence of interrupt-driven concurrency. It does this using a two-element lattice to conservatively detect *racing variables* that are accessed directly or indirectly by multiple flows (i.e., either by multiple interrupt handlers, or else by one interrupt handler and by the non-interrupt context) and whose accesses are not uniformly protected by disabling interrupts. Racing variables are considered to have the value \perp at all program points. Non-racing variables are always manipulated atomically and can be modeled soundly by identifying all program points where interrupts might become enabled, and then adding flow edges from these locations to the start of all interrupt handlers.

Dataflow analysis of racing variables can be performed using essentially the same technique, where each access to a racing variable is considered to be an atomic access, after which flow edges to interrupts must be added. We leave this feature in CComp turned off by default because in our experience it does not significantly increase precision and because it is potentially unsound when language-level memory operations are compiled down to non-atomic instruction sequences.

3.5 Integrated analyses

A well-known way to avoid phase-ordering problems in compilers is to create integrated super-analyses that run multiple analyses concurrently and permit them to exchange information with each other. For example, conditional constant propagation [31] integrates constant propagation and dead code detection. CComp integrates all of its main analyses: value set analysis, dead code detection, pointer set analysis, and concurrency analysis. Although we have not quantified the benefits of integrating these analyses, it seems clear that, at least in some cases, there was no reasonable alternative. For example, concurrency analysis cannot run before pointer analysis (in this case aliases cannot be tracked accurately, forcing a highly pessimistic race analysis) or after it (racing pointers will have been analyzed unsoundly).

3.6 Whole-program optimization

The results of CComp’s interprocedural dataflow analysis are used to perform interprocedural constant propagation, pointer optimizations, dead code elimination, dead data elimination, redundant synchronization elimination, copy propagation, and branch, switch, and jump optimizations.

3.7 Modeling volatile variables

The C99 standard states that:

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects.

In practice, volatile variables are not subject to most compiler optimizations. Similarly, early versions of CComp did not attempt to model volatiles, treating them as perpetually \perp .

Eventually, we realized not only that bottom-valued global variables were hurting CComp’s precision (they tend to be contagious) but also that many volatile variables can be analyzed soundly. The insight is that volatiles are only opaque at the language level. When non-portable platform-level properties are known, different uses of the volatile qualifier can be identified:

1. Storage locations corresponding to device registers are volatile in the most literal sense: their values can change at any time and both reads and writes may be side-effecting. Data flowing through these locations cannot be analyzed by CComp.
2. Global variables in regular RAM are often declared as volatile to prevent compiler-level caching of values. These variables cannot, in fact, be modified except by stores in the program—which may occur in interrupt handlers. C does not have a semantics for interrupts, but CComp does, and so it can safely analyze and even compress volatile global variables.

CComp identifies references to device registers heuristically, by matching against the two important idioms for accessing hardware devices from C code. The first method is portable: a constant integer is cast into a pointer-to-volatile and then dereferenced. The second method is non-portable: gcc permits variables to be bound to specific memory locations occupied by device registers. We claim that dataflow analysis through volatiles is sound under the assumption that our heuristic correctly identifies accesses to device registers.

3.8 Analyzing arrays

CComp represents each array using a single abstract value that tracks the greatest lower bound of any value stored in any element of the array. We had two reasons for choosing a collapsed array representation. First, it is often the case that the elements of a given array store similar values. For example, a common use of arrays in networked embedded applications is to store a queue of pointers to packet buffers. In this case, the pointer set stored in each element of the array is the same—the set of buffers—so our analysis loses nothing compared to a more precise array model. Second, collapsed arrays are efficient.

3.9 Scalar replacement of aggregates

CComp compresses values found in C structs by splitting them into scalars and then attempting to compress the scalars. We created a scalar replacement pass that avoids hazards such as address-taken structs and structs inside unions. It also takes care to retain call-by-value semantics for arrays formerly inside structs. In C, structs are passed by value while arrays—due to being treated like constant pointers—are passed by reference.

3.10 Emulating external functions

cXprop treated external functions (those whose code is unavailable at analysis time) as *safe* or *unsafe*. Functions are unsafe by default, meaning that they must be modeled conservatively: when called they kill all address-taken variables and furthermore they are assumed to call back into address-taken functions in the application. A *safe* function is one like `scanf` that affects only program state passed to it through pointers. CComp improves upon this analysis by adding two new categories of external functions. A *pure* function, such as `strlen`, has no side effects. An *interpreted* function has a user-defined effect on program state. For example, we interpret calls to `memset` in order to perform array initialization. We could almost never compress an array before supporting this idiom.

```
// compression function for 16-bit variables
unsigned char __f_16 (uint16_t * table,
                    uint16_t val)
{
    unsigned int index;
    for (index=0; ; index++) {
        if (pgm_read_word_near (table + index) == val)
            return index;
    }
}

// decompression function for 16-bit variables
uint16_t __finv_16 (uint16_t * table,
                  unsigned char index)
{
    return pgm_read_word_near (table + index);
}
```

Figure 4. Compression and decompression functions based on table lookup. The function `pgm_read_word_near` is an AVR primitive for accessing values from ROM.

4. Compressing RAM

This section describes the theory and practice of offline RAM compression, including some optimizations.

4.1 Theory

Let x be a variable in a given program that occupies n bits. Since the embedded systems under consideration use static memory allocation, we can speak of variables rather than more general memory objects. Let V_x be a conservative estimate of the set of values that can be stored into x across all possible executions. In the general case (i.e., in future implementations of offline RAM compression), V_x could be computed by a type-based analysis, by a constraint-based analysis, by exhaustive testing, etc., as opposed to being computed using value set or pointer set analysis.

Offline RAM compression can in principle be performed when $|V_x| < 2^n$. However, in practice it should be the case that $\lceil \log_2 |V_x| \rceil < n$. In other words, compressing x by itself should result in a savings of at least one bit.

Exploiting the restricted size of V_x may be difficult because it may not be easy to represent the actual values in V_x compactly. A general solution is to find another set C_x with the same cardinality as V_x , and also to find a function f_x that is a one-to-one and onto mapping from V_x to C_x . Then, f_x is a *compression function* and its inverse f_x^{-1} is a *decompression function* (one-to-one and onto functions can always be inverted).

4.2 Finding and implementing f_x and f_x^{-1}

For each compressed variable x , we find f_x and f_x^{-1} as follows:

1. If x is an integer type and $\forall y \in V_x, 0 \leq y < 2^{\lceil \log_2 |V_x| \rceil}$, then the trivial compression function $f_x(y) = y$ can be used. Across our set of benchmark applications, 65% of compressed variables fall into this case.
2. Otherwise, we let the elements of C_x be $0..|V_x| - 1$ and let f_x and f_x^{-1} be lookups in a *compression table* stored in ROM. The compression table is simply an array storing all members of V_x .

Figure 4 depicts our compression table lookup functions. Decompression uses the compressed value as an index into the compression table, and compression involves a linear scan of the compression table. Empirically, on the AVR MCUs that we use to evaluate offline RAM compression, linear scan is faster than binary search, on average, for value set sizes up to 19. Another argument

against binary search is that since CComp operates by source-to-source transformation, we are unable to order pointers by value.

As we did this work, other implementations of compression and decompression suggested themselves. For example, if x is an integer type and we can find a constant c such that $\forall y \in V_x, c \leq y < c + 2^{\lceil \log_2 |V_x| \rceil}$, then $f_x(y) = y - c$ is a valid compression function. However, we did not implement this because we observed few cases where it would help.

4.3 Program transformation

For each compressed variable x , CComp performs the following steps:

1. If x requires a compression table, allocate the table in ROM.
2. Convert x 's initializer, if any, into the compressed domain.
3. Allocate space for x in a global compressed struct as a bitfield of $\lceil \log_2 |V_x| \rceil$ bits.
4. Rewrite all loads and stores of x to access the compressed bitfield and go through compression and decompression functions.

CComp does not attempt to compress racing variables; this naturally falls out of the concurrency analysis described in Section 3.4 that treats racing variables as \perp at all program points. CComp also does not attempt to compress floating point values. This was a design simplification that we made based on the lack of floating point code in the MCU-based applications that we target.

4.4 An example

Figure 5 illustrates RAM compression using code from the standard TinyOS application BlinkTask. The TOSH_queue data structure shown in Figure 5(a) is at the core of the TinyOS *task scheduler*, which supports deferred function calls that help developers avoid placing lengthy computations in interrupt handlers. Elements of the task queue are 16-bit function pointers on the AVR architecture. Through static analysis, CComp shows that elements of the task queue have pointer sets of size four. The contents of this pointer set are used to generate the compression table shown in Figure 5(b). Since each element of the task queue can be represented using two bits, an 8:1 compression ratio is achieved, saving a total of 14 bytes. Figure 5(c) shows part of the compressed data region for the BlinkTask application. Finally, Figure 5(d) shows how the original application reads a function pointer from the task queue and Figure 5(e) shows the transformed code emitted by CComp.

4.5 A global synchronization optimization

Since compressed data is accessed through non-atomic bitfield operations, an unprotected global compressed data region is effectively a big racing variable. To avoid the overhead of disabling interrupts to protect accesses to compressed data, we instead decided to segregate the compressed data into two parts: one containing variables whose accesses are protected by disabling interrupts, the other containing variables that do not need to be protected because they are not accessed by concurrent flows. Neither of the segregated compressed data regions requires explicit synchronization.

4.6 A global layout optimization

As shown in Figure 5(c), we represent compressed RAM using C's bitfield construct. The cost of a bitfield access depends on the alignment and size of the field being accessed. There are three major cases for bitfield sizes of less than a word: a bitfield aligned on a word boundary, a bitfield that is unaligned and does not span multiple words, and a bitfield that is unaligned and spans words. Figure 6 summarizes the code size and performance costs of these different cases for our toolchain. Other architectures and compilers would

(a) Original declaration of the task queue data structure:

```
typedef struct {
    void (*tp)(void);
} TOSH_sched_entry_T;

volatile TOSH_sched_entry_T TOSH_queue[8];
```

(b) Compression table for the task queue (the progmem attribute places constant data into ROM):

```
unsigned short const __attribute__((__progmem__))
__valueset_3[4] = {
    NULL,
    &BlinkTaskM$processing,
    &TimerM$HandleFire,
    &TimerM$signalOneTimer
};
```

(c) The compressed task queue is element f9 of the global compressed data region, which has room for eight two-bit compressed values:

```
struct __compressed {
    char f9[2];
    unsigned char f0 : 3;
    unsigned char f1 : 3;
    unsigned char f7 : 1;
    ...
};
```

(d) Original code for reading the head of the task queue:

```
func = TOSH_queue[old_full].tp;
```

(e) Code for reading the head of the compressed queue (the "2" passed to the array read function indicates that compressed entries are two bits long):

```
__tmp = __array_read (__compressed.f9, old_full, 2);
func = __finv_16 (__valueset_3, __tmp);
```

Figure 5. Compression transformation example for the main TinyOS 1.x scheduler data structure: a FIFO queue of function pointers for deferred interrupt processing

access type	read		write	
	bytes	cycles	bytes	cycles
aligned	5	10	9	18
unaligned	7.2	14.4	11.2	22.4
spanning	13	26	21	42

Figure 6. Average bitfield access costs for the AVR processor and gcc

have different costs, but we would expect the ratios to be roughly the same since the unaligned and spanning cases fundamentally necessitate extra ALU and memory operations.

Rather than attempting to compute an optimal layout for compressed data, we developed an efficient greedy heuristic that attempts to meet the following two goals. First, the compressed variables with the most static accesses should be byte-aligned. On the AVR architecture, words are bytes. Second, no compressed variable should span multiple bytes. The heuristic operates as follows:

1. For each positive integer n less than the number of compressed variables:
 - (a) Align on byte boundaries the n variables with the largest number of static accesses.

- (b) Starting with the largest (compressed size) remaining variable, pack it into the structure under the constraint that it may not span two bytes. A variable is placed in the first location where it fits, extending the structure if it fits nowhere else. This step is repeated until there are no remaining variables to pack.
 - (c) Fail if there is any wasted space in the packed structure, otherwise succeed. Note that it is possible for there to be holes in the packed structure without wasting space. Instead of allowing the compiler-added padding at the end of a struct for alignment purposes, we disperse the padding throughout the struct.
2. Choose the succeeding result for the largest n .

In practice this heuristic works well, and in fact it almost always succeeds in byte-aligning the maximum possible number of high-priority compressed variables (that is, one per byte of compressed data). The heuristic can fail in the situation where it is forced to choose between wasting RAM and permitting a variable to span multiple bytes, but we have not yet seen this happen for any input that occurs in practice. If it does fail, we warn the user and back off to the unordered compressed structure.

4.7 Local optimizations

We implemented several additional optimizations. First, duplicate compression tables are eliminated, saving ROM. Second, when an application stores a constant into a compressed variable, the compression table lookup can be performed at compile time. `gcc` cannot perform this optimization on its own. Finally, we have `gcc` inline our compression and decompression functions. Since these functions are small, this gives a reasonable speedup with minimal code bloat.

5. Tradeoff-Aware RAM Compression

This section explores the idea of computing a cost/benefit ratio for each compressible variable and then selectively compressing only a highly profitable subset of the compressible variables. In other words, we give users a RAM compression knob to turn, where the 100% setting compresses all compressible variables, the 0% setting performs no compression, and at points in between the system attempts to give up as little ROM or CPU time as possible while achieving the specified level of compression. Tradeoff-aware compilation can help developers exploit the discontinuous nature of value functions for resource use (Section 1.1) by, for example, compiling an application such that it just barely fits into RAM, into ROM, or into an execution time budget.

Our RAM/ROM tradeoff is based on static information while our RAM/CPU tradeoff is based on profile information. We could have made the RAM/CPU tradeoff by using standard heuristics based on static information (e.g., “each loop executes 10 times”). However, we were reluctant to do so because in an interrupt-driven system, the dynamic execution count of each function is strongly dependent on the relative frequencies with which different interrupts fire.

The key to producing good tradeoffs is to accurately estimate the cost/benefit ratio of compressing each variable. We compute this ratio as follows:

$$\text{Cost/Benefit Ratio} = \frac{1}{S_u - S_c} \sum_{i=1}^6 C_i(A_i + B_i V) \quad (1)$$

S_u is original size of the compressible object, S_c is the compressed size, C is an *access profile*: a vector of the static or dynamic counts of the basic operations required to compress the variable, A and B are vectors of platform-specific, empirically determined cost pa-

access type	size cost (bytes)	speed cost (cycles)
bitfield read	6.1	12.2
bitfield write	10.1	20.2
array read	40	90
array write	60	120
decompress	24	16
compress	14	20 + 9.5V

Figure 7. Two sets of parameters that can be plugged into Equation 1 to support either trading RAM for ROM or RAM for CPU cycles

rameters, and V is the cardinality of the variable’s value set. Figure 7 shows two sets of parameters, one in terms of ROM cost and the other in terms of CPU cost, which we computed for the AVR processor. Most of the B constants are zero, meaning that operations have constant cost. The exception is the cycle cost to compress a value, which involves a linear scan of the compression table as shown in Figure 4. In some cases these costs are average figures across several datatypes. For example, since AVR is an 8-bit architecture, compressing a `char` is cheaper than compressing an `int`. We could have made this information more fine-grained by breaking out more sub-cases, but we judged that this had diminishing returns.

The static count of each basic compression and decompression operation naturally falls out of the compilation process. These counts are taken after the optimizations described in Section 4 have been performed. We measured dynamic operation counts by running applications in a modified version of the Avrora sensor network simulator [29].

6. Evaluation

We evaluate offline RAM compression by answering several questions: How much RAM is saved? What is the cost in terms of ROM usage and execution time? Are tradeoffs between resources effective? What is the cost in terms of analysis time?

Our evaluation is based on a collection of embedded applications for Atmel AVR 8-bit microcontrollers:

- Two robot control applications, Robot1 and Robot2, emitted by KESO [27], an ahead-of-time Java-to-C compiler for constrained embedded systems. They are 2526 and 3675 lines of code (LOC), respectively.
- An avionics control application, UAVcontrol, for a small unmanned aerial vehicle developed by the Paparazzi project [22] (4215 LOC).
- Eleven sensor networking applications from TinyOS [15] 1.1, emitted by the nesC [13] compiler (5800–39000 LOC).

In all cases our baseline is the RAM usage, ROM usage, and CPU usage of the out-of-the-box version of the application as compiled by `avr-gcc` version 3.4.3. For the TinyOS applications, we used Avrora [29], a cycle-accurate sensor network simulator, to measure efficiency in terms of *duty cycle*—the fraction of time the processor is active. This is a good metric because it directly correlates to energy usage and hence system lifetime: a primary consideration for sensor networks.

We did not measure the efficiency of the KESO or Paparazzi applications; they run on custom hardware platforms with many peripherals. Extending Avrora to simulate enough hardware that we could run these applications was beyond the scope of our work.

CComp does not change the semantics of an application unless it contains analysis or transformation bugs. To defend against this possibility, we validated many of our compressed applications. This

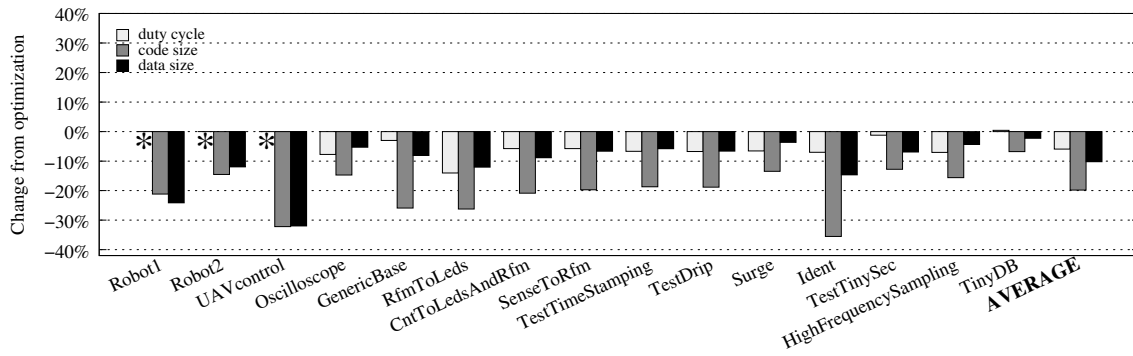


Figure 8. Change in resource usage of applications from whole-program optimization (Section 3.6), without RAM compression

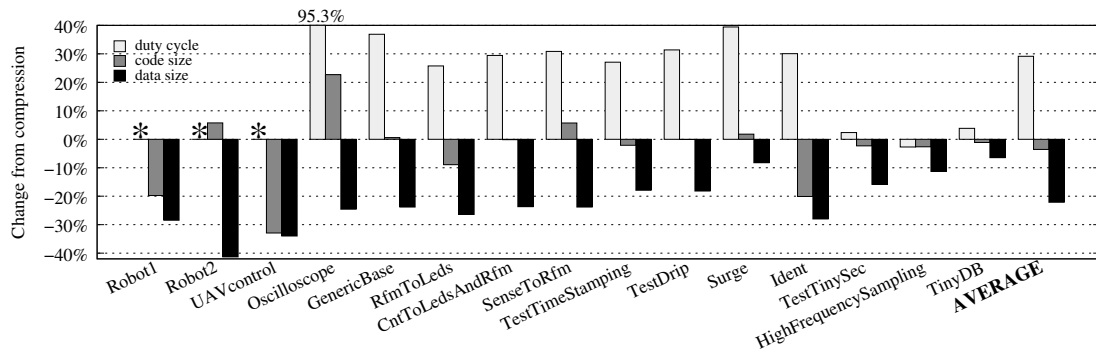


Figure 9. Change in resource usage of applications from whole-program optimization and maximum RAM compression

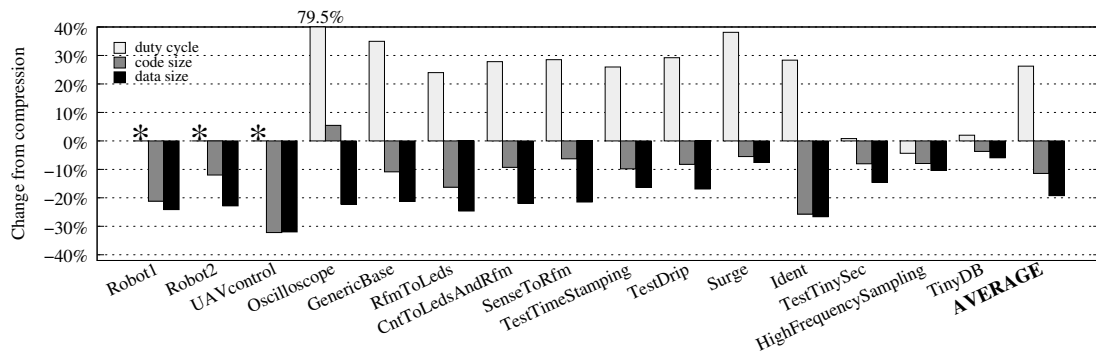


Figure 10. Change in resource usage of applications at the 90% compression level, trading RAM for code size

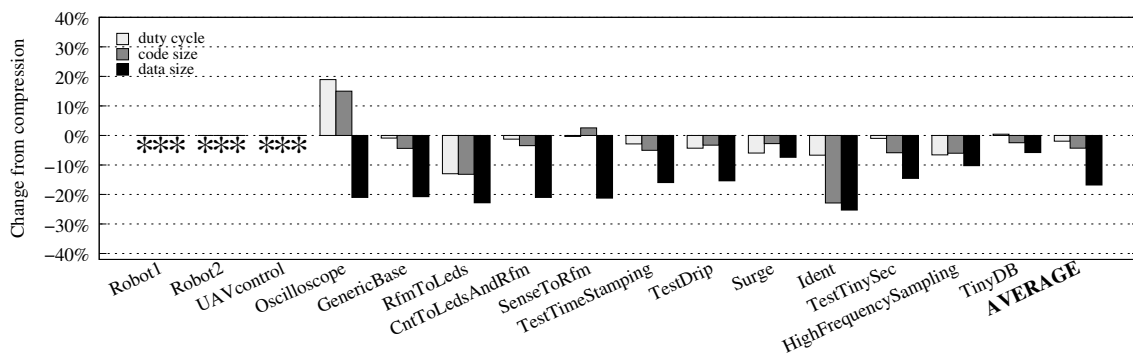


Figure 11. Change in resource usage of applications at the 90% compression level, trading RAM for duty cycle

was not totally straightforward because unlike benchmarks, real embedded applications do not simply transform inputs into outputs. Our validation was by hand, by checking that applications exhibited the expected behavior. For example, for Surge, a multihop routing application, we ensured that routes were formed and packets were properly forwarded across the network to the base station. We did not validate the applications for which we have no simulation environment: Robot1, Robot2, and UAVcontrol.

6.1 Effectiveness of optimizations

Section 4.2 mentions that across our benchmark suite, 65% of compressed variables meet our criteria for using simple compression and decompression functions, as opposed to indirection through a ROM-based compression table. On average, this optimization reduces code size by 11% and duty cycle by 22%. Intelligent layout of bitfields in the global compressed data structure (Section 4.6) reduces code size by 2.7% and duty cycle by 1.9%.

6.2 Per-application resource usage results

Figures 8–11 compare applications processed by CComp against a baseline of out-of-the-box applications in terms of code size, data size, and duty cycle. Asterisks in these figures mark data points that are unavailable because we cannot simulate some applications.

Figure 8 shows that when RAM compression is disabled, CComp reduces usage of all three resources through the optimizations described in Section 3.6. On the other hand, Figure 9 shows that maximum RAM compression results in significantly greater RAM savings: 22%, as opposed to 10% in Figure 8.

The problematic increase in average duty cycle shown in Figure 9 indicates that RAM compression can be expensive when hot variables are compressed. The general-purpose solution to this problem is tradeoff-aware compilation.

6.3 Results from tradeoffs

Figures 10 and 11 show the effect on our benchmark applications of turning the “RAM compression knob” down to 90%. That is, reducing RAM compression to 10% below the maximum in order to buy as much code size or duty cycle as possible. Figures 12 and 13 show the full spectrum of tradeoffs from 0% to 100% RAM compression. The most important thing to notice about these graphs is that sacrificing a small amount of RAM buys a major decrease in duty cycle and a significant decrease in code size. The steepness of these curves near 100% RAM compression indicates that our cost functions work well.

Figure 14 is a different way to look at the data from Figures 12 and 13. The diamond shape traced by the data points in this parametric plot provides additional validation that our tradeoff strategies are working properly.

6.4 Analysis time

The median time to apply offline RAM compression to members of our benchmark suite is 62 s. The minimum is 2 s and the maximum is 94 minutes. Only two applications (TestTinySec and TinyDB) require more than five minutes. CComp is a research prototype and we have not optimized it for speed.

7. Related Work

There is substantial body of literature on compiler-based RAM optimizations. Here we discuss representative publications from several categories of research that are related to ours. We do not discuss the literature on code compression and code size optimization, which is largely orthogonal to our work.

Compiler-based offline RAM size optimization. Ananian and Ri-nard [1] perform static bitwidth analysis and field packing for Java

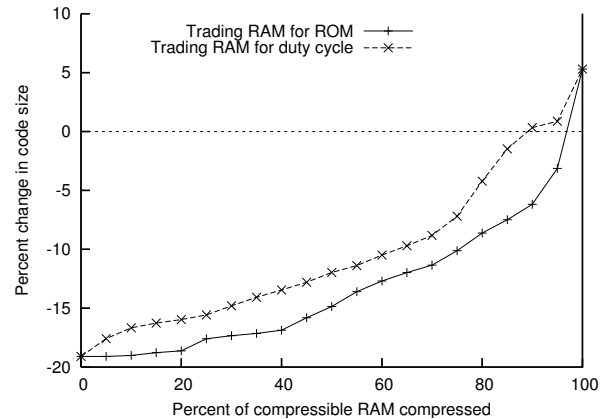


Figure 12. Quantifying the average effect on code size of turning the RAM compression knob from 0 to 100 while in trade-for-ROM and trade-for-duty-cycle modes

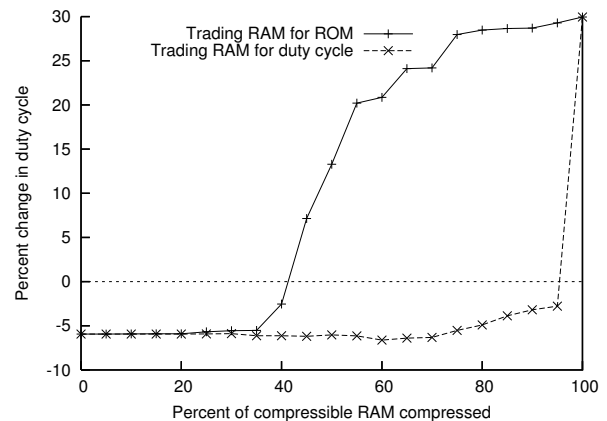


Figure 13. Quantifying the average effect on duty cycle of turning the RAM compression knob from 0 to 100 while in trade-for-ROM and trade-for-duty-cycle modes

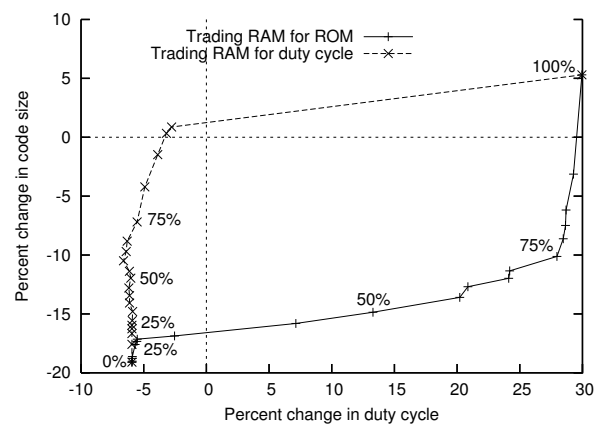


Figure 14. Parametric plot comparing CComp’s trade-for-ROM and trade-for-duty-cycle modes as the compression knob is turned from 0 to 100

objects (among other techniques less closely related to our work). Zhang and Gupta [33] use memory profile data to find limited-bitwidth heap data that can be packed into less space. Lattner and Adve [18] 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. [10] apply whole-program optimization to the Linux kernel at link time, reducing both RAM and ROM usage. Virgil [28] has a number of offline RAM optimizations including reachable members analysis, reference compression, and moving constant data into ROM.

Our work exploits the same basic insight as these previous efforts, but it differs in several key ways. First, we have taken a whole-system approach to compressing RAM for legacy C applications in the presence of interrupt-driven concurrency. Most previous work has focused on benchmarks rather than attacking actual embedded applications. Second, our value set and pointer set domains appear to work significantly better than do the interval and bitwise domains used to analyze bitwidth in most previous work. Third, we perform tight bit-level packing across multiple variables to achieve good savings on very small platforms. Fourth, we compress scalars, pointers, structures, and arrays. Most previous work has focused on some subset of these kinds of data. Finally, we have carefully focused on optimizations and tradeoffs to avoid slowing down and bloating applications unacceptably. This focus was necessary due to the constrained nature of the hardware platforms that we target and also because on cacheless systems there is no possibility of hiding overheads by improving locality.

A separate body of work performs offline RAM optimization using techniques that exploit the structure of runtime systems rather than (or in addition to) exploiting the structure of application data. Barthelmann [6] describes inter-task register allocation, a global optimization that saves RAM used to store thread contexts. Our previous work [25] addressed the problem of reducing stack memory requirements through selective function inlining and by restricting preemption relations that lead to large stacks. Grunwald and Neves [14] 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. We believe these techniques to be complementary to CComp: they exploit different underlying sources of savings than we do.

Static bitwidth analysis. A number of researchers, including Razdan and Smith [24], Stephenson et al. [26], Budiu and Goldstein [9], Ananian and Rinard [1], Verbrugge et al. [30], and ourselves [12] have developed compiler analyses to find unused parts of memory objects. Our research builds on this previous work, innovating in a few areas such as analyzing dataflow through volatile variables.

Online RAM size optimization. The constrained nature of RAM in embedded systems is well known and a number of research efforts have addressed this problem using online schemes that dynamically recover unused or poorly used space. Biswas et al. [7] and Middha et al. [19] 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 [32] 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 [2] provides on-line compression for systems without a memory management unit, such as wireless sensor network nodes. Ozturk et al. [21] compress data buffers in embedded applications. In contrast with our work, these techniques are opportunistic and not guaranteed to work well for any particular run of a system. Online RAM optimizations are most suitable for larger embedded platforms with RAM sized in hundreds of KB or MB where—

statistically—there are always enough opportunities for compression to provide good results. Also, most online RAM optimizations incur unpredictable execution time overheads (for example, uncompressing a memory page on demand) and therefore may not be applicable to real-time systems.

RAM layout optimizations. 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 [11] and Rabbah and Palem's work on data remapping [23]. This type of research relates mostly to our compressed structure layout work in Section 4.6. As far as we know, no existing work has addressed the problem of bitfield layout to minimize code size or execution time, as we have.

Value-set-based pointer analysis. Abstract values in our value set and pointer set domains are sets of explicit concrete values. Balakrishnan and Reps [3] used value set analysis to analyze pointers in executable code, but they use the term differently than we do. Their abstract values are sets of reduced interval congruences—a highly specialized domain tuned to match x86 addressing modes.

8. Conclusions

We developed a novel method for *offline RAM compression* in embedded software that employs static whole-program analysis in the value set and pointer set domains, followed by source-to-source transformation. We have shown that across a collection of embedded applications targeting small microcontrollers, compression reduces an application's RAM requirements by an average of 12%, in addition to a 10% savings through a dead data elimination pass that is also driven by our whole-program analysis. This result is significant because RAM is often the limiting resource for embedded software developers, and because the programs that we started with had already been tuned for memory efficiency. Our second main contribution is a tradeoff-aware compilation technique that, given a goal in terms of RAM savings, attempts to meet that goal while giving up as little code size or as few processor cycles as possible. Finally, we have created a tool, CComp, that implements offline RAM compression and tradeoff-aware compilation for embedded C programs.

Acknowledgments

We thank Eric Eide, Alastair Reid, and Bjorn De Sutter for their helpful comments on drafts of this paper. This work is supported by National Science Foundation CAREER Award CNS-0448047.

References

- [1] 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.
- [2] 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.
- [3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 5–23, Bonita Springs, FL, April 2004.
- [4] Ross Bannatyne. Microcontrollers for the automobile. *Micro Control Journal*, 2004. <http://www.mcjournal.com/articles/arc105/arc105.htm>.

- [5] Max Baron and Cheryl Cadden. Strong growth to continue for MCU market, 2005. <http://www.instat.com/press.asp?ID=1445&sku=IN0502457SI>.
- [6] 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.
- [7] 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.
- [8] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of the 5th Intl. Symp. on High Performance Computer Architecture (HPCA)*, Orlando, FL, January 1999.
- [9] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proc. of the European Conf. on Parallel Processing (EUROPAR)*, München, Germany, August 2000.
- [10] 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.
- [11] 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.
- [12] Nathan Coopridge and John Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 44–53, Ottawa, Canada, June 2006.
- [13] 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 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.
- [14] 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.
- [15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer 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, November 2000.
- [16] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 81–94, 2004.
- [17] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: A link layer security architecture for wireless sensor networks. In *Proc. of the 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, November 2004.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] The Paparazzi project, 2006. <http://www.nongnu.org/paparazzi>.
- [23] 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.
- [24] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the 27th Intl. Symp. on Microarchitecture (MICRO)*, pages 172–180, San Jose, CA, November 1994.
- [25] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, November 2005.
- [26] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.
- [27] Michael Stalkerich, Christian Wawersich, Wolfgang Schröder-Preikschat, Andreas Gal, and Michael Franz. An OSEK/VDX API for Java. In *Proc. of the 3rd Workshop on Programming Languages and Operating Systems (PLOS)*, San Jose, CA, October 2006.
- [28] 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)*, Portland, OR, October 2006.
- [29] 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)*, Los Angeles, CA, April 2005.
- [30] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation: A study in C. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, Linköping, Sweden, April 1996.
- [31] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991.
- [32] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. On-line memory compression for embedded systems. *ACM Trans. Embedded Computing Systems (TECS)*, 2006.
- [33] Youtao Zhang and Rajiv Gupta. Compressing heap data for improved memory performance. *Software—Practice and Experience*, 36(10):1081–1111, August 2006.