Offline Compression for On-Chip RAM

Nathan Cooprider and John Regehr
Software Tools for Sensor Networks
School of Computing
University of Utah
(coop@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. Our analysis and transformation 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. For a collection of embedded applications for AVR microcontrollers, our transformation reduces RAM usage by an average of 22%, of which 10% is due to dead-data elimination and 12% is due to compression. In addition, we developed a model for estimating the cost/benefit ratio of compressing each variable. This model supports a flexible spectrum of resource tradeoffs through selective compression of variables that present good value propositions.

On-Chip RAM

On-chip RAM is persistently scarce in low-end systems. RAM is— and always has been—in short supply in small embedded systems that are constrained by power, size, and unit cost.

Device  ROM  RAM  Ratio  Price
ATtiny24  2 KB  128 B  16:1  $0.70
ATtiny45  4 KB  256 B  16:1  $0.97
ATmega18  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 costs of some members of Atmel’s AVR family: a popular line of 8-bit MCUs. Prices are from http://digikey.com and other sources, in quantities of 100 or more.

Static Analysis
We use an agnostic static analysis in order to compute the set of values that can be stored in variables across all possible executions and to perform whole-program optimization.

$x$ = a variable
$n$ = number of bits $x$ occupies
$V$ = conservative estimate of the set of values that can be stored into $x$ across all possible executions

If $|V| < 2^n$ then offline RAM compression can be performed in principle. If $\log_2 |V| < n$ then offline RAM compression can be performed in practice (saves at least one bit).

![Resource usage of applications after whole-program optimization without RAM compression](image)

Offline Compression

Compression and decompression functions based on table lookup. The function pgm read word near is an AVR primitive for accessing values from ROM.

// compression function for 16-bit variables
unsigned char __compress_16 (uint16_t * table, uint16_t val) {
  unsigned int index = 0;
  while (1) {
    if (pgm_read_word_near (table + index) == val)
      return index;
    index++;
  }
}

// decompression function for 16-bit variables
uint16_t __decompress_16 (uint16_t * table, unsigned char index) {
  return pgm_read_word_near (table + index);
}

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

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

typedef struct {
  void (*tp)(void);
  TOSH_sched_entry_T
} TOSH_queue[8];

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

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

(c) The compressed task queue is element $f_0$ of the global compressed data region, which has room for eight two-bit compressed values:

struct __compressed {
  char $f_9[2]$ ;
  unsigned char $f_0$ : 3 ;
  unsigned char $f_1$ : 3 ;
  unsigned char $f_7$ : 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 “$a$” passed to the array read function indicates that compressed entries are two bits long):

__tmp = __array_read(__compressed.f9, old_full, 2);

![Evaluation](image)

Evaluation

![Source usage of applications after whole-program optimization and maximum RAM compression](image)

Figure 4. Resource usage of applications after whole-program optimization and maximum RAM compression

Cost Benefit Ratio = $\frac{1}{(S - S_0)} \sum \frac{1}{(A_i + B_i)}$

Figure 5. Formula for computing ordering of compression for tradeoffs

Figure 6. 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 7. Quantifying the average effect on duty cycle of turning the “RAM compression knob” from 0 to 100 while in trade-for-RAM and trade-for-duty-cycle modes

Conclusion

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, we can reduce static RAM requirements by 22%, on average. 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

This work is supported by National Science Foundation CAREER Award CNS-0448047

http://www.cs.utah.edu/~coop/research/ccomp