Offline Compression for On-Chip RAM

John Regehr
University of Utah

Joint work with Nathan Cooprider
Microcontrollers (MCUs)

- $12.5$ billion market in 2006
- ~10 billion units / year
- MCU-based products typically highly sensitive to unit cost
On-Chip RAM is Small

- Atmel AVR examples
  - mega48 – 0.5 KB RAM – $1.50
  - mega128 – 4 KB RAM – $8.75
  - mega256 – 8 KB RAM – $10.66

- SRAM can dominate power consumption of a sleeping chip

- On-chip flash memory is larger
  - 1 transistor / bit vs. 6 transistors / bit for SRAM
Out of RAM – What Next?

- Manually optimize RAM usage?
- Remove features?
- Buy MCUs with more RAM?
Is RAM Used Efficiently?

- Performed byte-level value profiling for TinyOS applications
  - Simulator counts distinct values stored in each byte of RAM over a number of executions

- Result: Average byte stores four values!
  - These applications are already tuned for RAM usage
RAM Compression

1. Use static analysis to find redundancy in scalars, pointers, structs, arrays in embedded C code

2. Automatically perform sub-word packing to save RAM
Static Analysis

- Input: C program
  - For example, output of nesC compiler
- Output: Value set for each variable
  - E.g. \{0,1\} for a boolean
- Analysis built on standard techniques
- However:
  - Very aggressive compared to e.g. gcc
  - Many tricks to get good precision on TinyOS applications – several novel
\[ x \overset{\text{def}}{=} \text{variable that occupies } n \text{ bits} \]

\[ V_x \overset{\text{def}}{=} \text{conservative estimate of value set} \]

\[ \lceil \log_2 |V_x| \rceil < n \implies \text{RAM compression possible} \]

\[ C_x \overset{\text{def}}{=} \text{another set such that } |C_x| = |V_x| \]

\[ f_x \overset{\text{def}}{=} \text{bijection from } V_x \text{ to } C_x \]

\[ n - \lceil \log_2 |C_x| \rceil \implies \text{bits saved through compression of } x \]
void (*function_queue[8])(void);
\[ n = \text{size of a function pointer} = 16 \text{ bits} \]

```c
void (*function_queue[8])(void);
```
$n = 16 \text{ bits}$

$|V_x| = 4$

$\left\lfloor \log_2 |V_x| \right\rfloor < n$

$2 < 16$
\[ f_x \overset{\text{def}}{=} V_x \text{ to } C_x \overset{\text{def}}{=} \text{compression} \]
\[ f_x^{-1} \overset{\text{def}}{=} C_x \text{ to } V_x \overset{\text{def}}{=} \text{decompression} \]
\[ V_x = \{ \text{cloud1}, \text{cloud2}, \text{cloud3}, \text{cloud4} \} \]

\[ f_x \overset{\text{def}}{=} \text{compression table scan} \]

\[ f_x^{-1} \overset{\text{def}}{=} \text{decompression table lookup} \]
$V_x = \{ \text{clouds}, \text{clouds}, \text{clouds}, \text{no symbol} \}$

128 bits reduced to 16 bits
112 bits of RAM saved
Implementation

- Source-to-source transformation for C
  - Rewrite declaration, initializer, reads, writes
- What about value sets of size 1?
  - $\log_2 1 = 0$
  - No bits required – just propagate the constant
- Optimizations
  - Avoid table-driven compression funcs when possible
  - Align hot compressed values on word boundaries
  - Merge redundant compression tables
  - Compile-time compression when storing constants
RAM Compression Results

- **Robot2**: Duty cycle decrease by 40%, code size decrease by 30%, data size decrease by 20%
- **GenericBase**: Duty cycle decrease by 10%, code size decrease by 5%, data size decrease by 2%
- **CntToLedsAndRfm**: Duty cycle decrease by 20%, code size decrease by 15%, data size decrease by 10%
- **Ident**: Duty cycle decrease by 15%, code size decrease by 10%, data size decrease by 5%
- **TinyDB**: Duty cycle increase by 5%, code size increase by 10%, data size increase by 15%

*⇒ simulator unavailable
RAM Compression Results

Constant Prop / DCE
10% RAM reduction
20% ROM reduction
5.9% duty cycle reduction

⇒ simulator unavailable
RAM Compression Results

Constant Prop / DCE
10% RAM reduction
20% ROM reduction
5.9% duty cycle reduction

Compression
22% RAM reduction
3.6% ROM reduction
29% duty cycle increase

⇒ simulator unavailable
Tuning RAM Compression

- Can elect to not compress some compressible variables
  - But which ones?
- For each compressible variable compute a cost / benefit ratio
  - Cost – estimated penalty in ROM or CPU cycles
  - Benefit – RAM savings
- Sort compressible variables by ratio and compress until some threshold is met
Cost/Benefit Ratio

\[ \sum C_i (A_i + B_i V) \]

- \( C \) \( \equiv \) access profile
- \( A, B \) \( \equiv \) platform-specific costs
- \( V \) \( \equiv \) cardinality of value set
Cost/Benefit Ratio

\[ \sum C_i (A_i + B_i V) \]

* \( C \) ≡ access profile
* \( A, B \) ≡ platform-specific costs
* \( V \) ≡ cardinality of value set

\[ S_u - S_c \]

* \( S_u \) ≡ original size
* \( S_c \) ≡ compressed size
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle

code size

data size

0%
Turning the RAM Knob

The graph illustrates the percent change in duty cycle, code size, and data size as a function of the percent of compressible RAM compressed. The arrow indicates a 10% compression.
Turning the RAM Knob

Duty cycle: 
Code size: 
Data size: 

20%
Turning the RAM Knob

![Graph showing the effect of turning the RAM knob on duty cycle, code size, and data size. The graph indicates a 30% change in the data size.]

- Duty cycle
- Code size
- Data size

Percent of compressible RAM compressed

Percent change
Turning the RAM Knob

The graph shows the percent change in duty cycle, code size, and data size as a function of the percent of compressible RAM that is compressed. The graph indicates a significant decrease in all three parameters as the compression level increases, with a 40% compression showing a notable effect.
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle
code size
data size

50%
Turning the RAM Knob

![Graph showing the percent change in duty cycle, code size, and data size against the percent of compressible RAM that is compressed. The graph indicates a 70% reduction in size at certain points.](image-url)
Turning the RAM Knob

- Duty cycle
- Code size
- Data size

Percent change vs. Percent of compressible RAM compressed
Turning the RAM Knob

![Graph showing the percent change in duty cycle, code size, and data size against the percent of compressible RAM compressed. The graph indicates a 90% improvement.](image)
Turning the RAM Knob

Graph showing the percent change in duty cycle, code size, and data size against the percent of compressible RAM compressed.
Turning the RAM Knob

![Graph showing percent change in duty cycle, code size, and data size against percent of compressible RAM compressed. The graph indicates 95% efficiency.]

Percent change

Percent of compressible RAM compressed
Compression Spectrum

Trading RAM for duty cycle

Percent change in code size

Percent change in duty cycle

-20 -15 -10 -5 0 5 10 15 20 25 30 35

0% 25% 50% 75% 95% 100%
Conclusion

- RAM likely to remain scarce in low-cost, low-power MCUs
- RAM is used inefficiently
- Manually RAM optimization is unpleasant
- Tradeoffs are useful

- CComp implements RAM compression
## Analysis Times

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HH:MM:SS</th>
<th>Benchmark</th>
<th>HH:MM:SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>drive1</td>
<td>00:00:13</td>
<td>sensetorfm</td>
<td>00:01:42</td>
</tr>
<tr>
<td>drive2</td>
<td>00:00:22</td>
<td>testtimestamping</td>
<td>00:02:09</td>
</tr>
<tr>
<td>osc</td>
<td>00:00:44</td>
<td>ident</td>
<td>00:03:13</td>
</tr>
<tr>
<td>genericbase</td>
<td>00:01:17</td>
<td>surge</td>
<td>00:03:42</td>
</tr>
<tr>
<td>ap</td>
<td>00:01:18</td>
<td>hfs</td>
<td>00:06:06</td>
</tr>
<tr>
<td>rfmtoleds</td>
<td>00:01:23</td>
<td>testtinysec</td>
<td>00:12:34</td>
</tr>
<tr>
<td>cnttoledsandrfm</td>
<td>00:01:32</td>
<td>tinydb</td>
<td>01:58:45</td>
</tr>
<tr>
<td>testdrip</td>
<td>00:01:42</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
No RAM Compression

![Bar chart showing change from optimization for different benchmarks.]
Full RAM Compression

- **Robot2**: Duty cycle significantly increased.
- **GenericBase**: Major decrease in code size.
- **CntToLedsAndRfm**: Minuscule changes in data size.
- **Ident**: Minimal impact on code size.
- **TinyDB**: Moderate decrease in data size.

*⇒ simulator unavailable