Removing the Memory Limitations of Sensor Networks with Flash-Based Virtual Memory

Andreas Lachenmann, Pedro José Marrón, Matthias Gauger, Daniel Minder, Olga Saukh, Kurt Rothermel

Universität Stuttgart, Germany
ACM SIGOPS, 2007
Outline

- Introduction
- System Properties and Design
  - Sensor Network Characteristics
  - Design Goals
- Memory Layout Algorithm
- Implementation
- Evaluation
- Conclusions
Introduction (1/3)

- RAM limitations
  - Main memory is very scarce (MICA2 : 4kB)
  - Sensor network applications require more memory
    - TinyDB
    - Maté

- OSs for sensor networks except for t-kernel don’t support virtual memory
  - T-Kernel v.s. TinyOS (OS)
  - T-Engine v.s. MICA family (hardware)
Introduction (2/3)

- **Flash memory**
  - The overhead of data accessing to flash memory
    - Accessing time
    - Energy consumption

- **TinyOS is suited for virtual memory**
  - variables, addresses are known at compile-time
    - except for stack
  - the locality of reference
ViMem

- provide a virtual memory abstraction for TinyOS-based sensor networks
- use flash memory to extend the size of RAM available to the application
- minimize the number of flash memory operations
  - use *variable access traces* to rearrange variables in virtual memory at *compile-time*
Sensor Network Characteristics

- ViMem has to be implemented in software
- the behavior of flash memory
  - Accessing time: writing / reading $\approx 4.5$
  - Energy consumption: writing / reading $\approx 23$
- wear leveling

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>264 bytes</td>
</tr>
<tr>
<td>Number of pages</td>
<td>2048</td>
</tr>
<tr>
<td>Number of internal SRAM buffers for pages</td>
<td>2</td>
</tr>
<tr>
<td>Max. standby current</td>
<td>10 $\mu$A</td>
</tr>
<tr>
<td>Max. page read current</td>
<td>10 mA</td>
</tr>
<tr>
<td>Max. page write current</td>
<td>35 mA</td>
</tr>
<tr>
<td>Typical page read delay (measured)</td>
<td>3.6 ms</td>
</tr>
<tr>
<td>Typical page write delay (measured)</td>
<td>16.3 ms</td>
</tr>
</tbody>
</table>

Table 1: Properties of the Atmel AT45DB041B flash memory chip
Design Goals for ViMem
- ViMem should not require hardware support
- ViMem minimize the number of write accesses
- ViMem should be efficient for frequently used variables
- Control which variables placed in virtual memory
- Accesses to variables in virtual memory are transparent
- Reuse of existing application and system components
ViMem consists of two main parts:

- compiler extension
  - Redirects variable accesses to ViMem’s runtime system
  - Determines the placement of variables on the memory pages
    - pre-compiler executes a memory layout algorithm
ViMem consists of two main parts:

- runtime component
  - loading and storing flash memory pages
    - Enhanced Second-Chance Algorithm
  - makes use of the SRAM buffers

- The limit of writing pages to flash
  - Each flash page can be written a fixed number of times
  - A pool of flash memory pages
Memory Layout Algorithm (1/3)

- The memory layout heuristic
  - determines the placement of variables in virtual memory
  - Use of Variable Access Traces
    - if the exact order of accesses were known at compile-time
      => NP-complete problem
    - test the application and to obtain a data access trace
    - pre-compiler uses the number of variable references in the source code
    - pre-compiler splits up complex variables
Memory Layout Algorithm (2/3)

- Grouping of Data Elements
  - $G = (V, E, f(v), g(e))$

![Diagram of Memory Layout Algorithm](image)

Access trace:
- Read s.y
- Write a
- Read a
- Write s.y
- Read s.x
- Read b
- Read a
- ...

Recently accessed:
- s.y: 2 bytes
- a: 1 byte
- s.y: 2 bytes
- s.x: 4 bytes
- b: 2 bytes
- a: 1 byte

Max. 8 bytes

Figure 2: Example for processing an access trace
Memory Layout Algorithm (3/3)

Data Placement

\[ p = \frac{g(e)}{f(v1) + f(v2)} \]

Figure 3: Example for the memory layout algorithm
Implementation (1/5)

- OS: TinyOS
- hardware: MICA family of sensor nodes
- compiler: nesC
- pre-compiler: JavaCC as a parser generator
- simulator: MICA2 simulator Avrora
- the softwares for test: TinyDB, Maté
Implementation (2/5)

Figure 4: ViMem compilation process
Implementation (3/5)

- **variables**
  - in virtual memory: declared globally or static
  - in RAM on the stack: local variables of functions
  - ID of a data element: stored in an array in program memory
  - @vm, @vmptr

```
1 uint16_t varInVM @vm();
2 uint16_t* pointer @vmptr();
3 uint16_t varInRAM;
4
5 uint16_t* testFunction()
6   uint16_t* value @vmptr() @vmptr() {
7       *value = 54;
8       return value;
9   }
10
11 command result_t StdControl.init() {
12   varInRAM = 123;
13   varInVM = varInRAM;
14   pointer = &varInVM;
15   varInVM = *testFunction(pointer);
16   return SUCCESS;
17 }
```

Figure 5: nesC code that accesses variables stored in virtual memory
Implementation (4/5)

- translate address of an ID of a data element
  - ID is known at compile-time:
    - most kinds of variables
      - pre-compiler inserts a direct call to the runtime system
  - ID is not known at compile-time:
    - pointers and A[b]
      - runtime system look up the page and offset
Implementation (5/5)

- Runtime Component
  - modified TinyOS PageEEPROM component
    - the flash component resets its state immediately after executing a command
  - Reading flash pages without buffers
  - Using buffers as caches for modified pages for reading again

<table>
<thead>
<tr>
<th>Type of access</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable in RAM</td>
<td>1.09 μs</td>
</tr>
<tr>
<td>VM variable in RAM</td>
<td>18.72 μs</td>
</tr>
<tr>
<td>VM var. from buffer without page write</td>
<td>3.66 ms</td>
</tr>
<tr>
<td>VM var. from flash without page write</td>
<td>3.72 ms</td>
</tr>
<tr>
<td>VM var. from flash with page write to buffer</td>
<td>7.58 ms</td>
</tr>
<tr>
<td>VM var. from flash with page write to flash</td>
<td>19.83 ms</td>
</tr>
</tbody>
</table>

Table 2: Typical latencies for different kinds of variable accesses
Evaluation (1/6)

- Isolated Memory Access Performance
  - the transfer time of the page from the flash memory chip to the CPU
    - reduce the number of flash accesses
    - a small number of variables@vm outside RAM
  - access to virtual memory is only allowed in non-time-critical functions
  - experience in other domains
Evaluation (2/6)

- Application Performance
  - applications: TinyDB, Maté
    - RAM-intensive
    - available in the TinyOS CVS repository
    - keep just one memory page (264 bytes) in RAM
    - performance is just influenced by the memory layout
  - test for 1000 seconds

<table>
<thead>
<tr>
<th>Application</th>
<th>Number</th>
<th>Size</th>
<th>Pages used</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyDB</td>
<td>75</td>
<td>569 bytes</td>
<td>2.15</td>
</tr>
<tr>
<td>Maté</td>
<td>1</td>
<td>792 bytes</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 3: Variables moved to virtual memory

<table>
<thead>
<tr>
<th>Application</th>
<th>Original</th>
<th>ViMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyDB</td>
<td>2,832 bytes</td>
<td>2,577 bytes</td>
</tr>
<tr>
<td>Maté</td>
<td>3,196 bytes</td>
<td>2,727 bytes</td>
</tr>
</tbody>
</table>

Table 4: Allocated space in RAM
Evaluation (3/6)

- Simulation of TinyDB
  - 90% of all accesses refer to just 20% of the bytes in virtual memory

Figure 6: Simulation of TinyDB
Evaluation (4/6)

- Simulation of Maté
  - less than 6 % of the data elements allocated in virtual memory are used in 90 % of all accesses
  - 10 faults / 38 accesses for initialization

Figure 7: Simulation of Maté
Large Data Storage: TinyDB
- (5,000 data accesses) * 10
- 3168 bytes in virtual memory
- trace without queries
- 1000 seconds

Figure 8: Varying the number of pages in RAM
Evaluation (6/6)

- Large Data Storage: TinyDB
  - (5,000 data accesses) * 10
  - 15840 bytes in virtual memory
  - Trace without queries
  - 1000 seconds

Figure 9: Varying the size of the data in virtual memory
Conclusions

- ViMem
  - does not require special hardware support for virtual memory
  - using ViMem is simple for developers
  - layout based on data access traces obtained from simulation or the source code itself
  - the increase of energy consumption of ViMem can be almost neglected