Making Life a Little Easier or...
Compile Time Checking of Run Time Resource Management

Presented by Roy Shea
With help from Shane Markstrum,
Todd Millstein, Rupak Majumdar, and Mani Srivastava
Making Life a Little Easier...

- Developing applications for sensor networks
  - Requires writing code
  - Low layer access to hardware
  - Limited insight into system
- Need to provide support for the developer

Back when I was a stressed programmer.
Focus on Resource Management

- Tool that examines source code to help programmers
  - Locate memory leaks in applications
  - Warn about dereferencing a dangling pointer
- Important on sensor node platforms
  - Memory leak rapidly exhausts the limited memory available
  - No MMU protection so following a dangling pointer can crash your node or worse...
Outline

- Overview of resources in sensor networks
- Model for proper resource usage
- A tool to automate verification
- Evaluation of verification
- Closing Remarks
...or Compile Time Checking of Run Time Resource Management

- Resource management
  - An item of limited quantity
  - One or more consumers
  - Often tied to hardware constraints of memory, buses, or sensors

- Run time
  - While an application is running (on real hardware)
  - Late feedback to developer

- Compile time checking
  - Early feedback
  - Limited visibility

```
  typeCheck.c: In function `main':
  typeCheck.c:11: warning: passing arg 1 of `add' makes integer from pointer without a cast
```
Resource Usage in Sensor Networks

- Buffer pools in TinyOS extensions such as VanGo
- Pointer swapping in TinyOS communication stack
- Dynamic memory in SOS
- Emerging resources
  - Access to hardware buses
  - Access to physical sensors

CENS Seminar Series

April 21, 2006
Outline

- Overview of resources in sensor networks
- Model for proper resource usage
- A tool to automate verification
- Evaluation of verification
- Closing Remarks
A Shortage of Laptops

• Limited number of laptops for too many graduate students

• Many potential problems
  – Grab a laptop that some is all ready using
  – Forget to return laptop
  – “But I thought you would grab it...”
Intuitive Resource Model

• A laptop should only be under the control of one student at any given time
• The student controlling the laptop must either keep track of the laptop, return the laptop, or give the laptop to another student
Intuitive Resource Model

- A laptop Memory should only be under the control of one student module at any given time.
- The student module controlling the laptop memory must either keep track of the laptop memory, return the laptop memory or give the laptop memory to another student module.
Resource Management API from SOS

• Memory ownership in SOS
  – Module allocating data owns that data
  – API exists to transfer ownership to another module

• Allocating and freeing memory
  – void * sys_malloc (uint16_t size)
  – void sys_free (void *ptr)

• Transferring and taking memory ownership
  – int8_t sys_post (..., uint8_t size, void *data, SOS_MSG_RELEASE)
  – void * sys_msg_take_data (void *data)
Sample Application: Surge

- Nodes sense data and send to sink
- Representative of many protocol stack based sensor network applications
  - Sensor data
  - Routing tree
  - Sink application
int8_t surge_module(void *state, Message *msg) {
    surge_state_t *s = (surge_state_t*)state;
    switch (msg->type) {
        case MSG_DATA_READY: {
            SurgeMsg* pkt = (uint8_t*)sys_malloc(sizeof(SurgeMsg));
            if (pkt == NULL) break;
            pkt->data = ... ; // ... set up message
            sys_post(..., sizeof(SurgeMsg), (void*)pkt, SOS_MSG_RELEASE);
            break;
        }
        case MSG_TR_DATA_PKT: {
            if (sys_id() == SURGE_BASE_STATION_ADDRESS) {
                uint8_t *payload = sys_msg_take_data(msg);
                sys_post(..., msg->len, payload, SOS_MSG_RELEASE);
                return SOS_OK;
            }
        }
        break;
    }
    return SOS_OK;
}
Formal Model

• Easily described using Linear Temporal Logic

\[ \square (\text{alloc } x) \rightarrow \Diamond (\text{free } y \lor \text{store } y) \land \text{alias}(x, y) \]

• No Memory Leaks

\[ \square (\text{free } x \rightarrow \bigcirc (\square (\neg (\text{access}(y) \land \text{alias}(x, y)))))) \]

• No Dangling Pointers

<table>
<thead>
<tr>
<th>LTL Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\square f$</td>
<td>holds at state $s$ if the formula $f$ holds on $s$ and each subsequent state in the path</td>
</tr>
<tr>
<td>$\Diamond f$</td>
<td>holds at state $s$ if the formula $f$ holds on $s$ or some subsequent state in the path</td>
</tr>
<tr>
<td>$\bigcirc f$</td>
<td>holds at state $s$ if the formula $f$ holds on the successor state of $s$ along the path</td>
</tr>
</tbody>
</table>
Outline

- Overview of resources in sensor networks
- Model for proper resource usage
- A tool to automate verification
- Evaluation of verification
- Closing Remarks
Automated Verification

- C Intermediate Language (CIL) written in OCaml
- Dataflow analysis
- Alias analysis
Preparing Code for Verification

- Annotations added to code describe memory manipulations
- Allocating resources
  - `void* __attribute__((sos_claim)) sys_malloc(int size);`
- Releasing resources
  - `int *state __attribute__((sos_store));`
  - `void sys_free(void *ptr __attribute__((sos_release)));
  - `sys_post` uses a conditional `sos_may_release` annotation with extra checker support
Sample Output

- **Memory Leaks**

  ```
  make check
  Warning: Allocated data from instruction #line 7 "oneFish.c"
  p = (int *)sys_malloc(__cil_tmp2);
  is not stored
  ```

- **Reference to dangling pointer**

  ```
  make check
  Warning: Var in #line 12 "twoFish.c"
  q = (*p);
  should be treated as dead
  
  Warning: Potential access to dead data freed in instruction #line 10
  free(__cil_tmp4);
  ```
Limitations

- Require some annotations for verification
  - Base set of annotations can be used to help “seed” iterative annotation of code
  - Small set of annotations goes a long ways
- Alias analysis has limited accuracy due to complex pointer manipulations in C
  - Developed tool can use external alias analysis that can be upgraded
- Coders write crazy code!
Outline

- Overview of resources in sensor networks
- Model for proper resource usage
- A tool to automate verification
- Evaluation of verification
- Closing Remarks
Dynamic Memory Usage in SOS

<table>
<thead>
<tr>
<th>Function</th>
<th>Total uses out of 46371 SLOC</th>
<th>Function</th>
<th>Total uses out of 12990 SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_malloc</td>
<td>135</td>
<td>sys_malloc</td>
<td>68</td>
</tr>
<tr>
<td>sys_msg_take_data</td>
<td>25</td>
<td>sys_msg_take_data</td>
<td>23</td>
</tr>
<tr>
<td>sys_free</td>
<td>124</td>
<td>sys_free</td>
<td>46</td>
</tr>
<tr>
<td>SOS_MSG_RELEASE</td>
<td>116</td>
<td>SOS_MSG_RELEASE</td>
<td>66</td>
</tr>
<tr>
<td>Other common memory operations</td>
<td>54</td>
<td>Other common memory operations</td>
<td>8</td>
</tr>
</tbody>
</table>

- Memory manipulation in one in 100 lines of code
- Mail archive reveals prominent threads on memory usage
- Common problem with new users
User Modules In SOS

- Pieces of applications similar to “components” in TinyOS
- CVS head includes 37 modules (April 2006)
  - Few simple applications
  - Many drivers to test aspects of the OS
- Combined these modules have 207 historic versions
- Verified all historic versions using a base set of 20 attribute annotations
Module Verification Results

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual memory leaks</td>
<td>16</td>
</tr>
<tr>
<td>Missing annotations</td>
<td>153</td>
</tr>
<tr>
<td>Free within a loop</td>
<td>66</td>
</tr>
<tr>
<td>False positives</td>
<td>72</td>
</tr>
</tbody>
</table>

- False positives...
  - Only used a fixed set of 20 annotations despite evaluating historical module versions (API drift)
  - Analysis is unable to reason about data freed in a loop
  - Complex data structures confuse the alias analysis
mod_op = (sos_module_op_t*) sys_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
if(mod_op->op == MODULE_OP_INSMOD) {
    existing_module = sys_get_module(mod_op->mod_id);
    if(existing_module != NULL) {
        uint8_t ver = sos_read_header_byte(
            existing_module->header, offsetof(mod_header_t, version));
        if (ver < mod_op->version) {
            sys_unload_module(existing_module->pid,
                sos_read_header_byte(existing_module->header,
                    offsetof(mod_header_t, version)));
        } else {
            return SOS_OK;
        }
    }
    ret = fetcher_request(sys_DFT_LOADER_PID, mod_op->mod_id,
        mod_op->version, entohs(mod_op->size), msg->saddr);
    s->pend = mod_op;
    sys_led(LED_RED_TOGGLE);
    return SOS_OK;
}
return SOS_OK;
Memory Leak in Module Loader (1)

```c
mod_op = (sos_module_op_t*) sys_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
if(mod_op->op == MODULE_OP_INSMOD) {
    existing_module = sys_get_module(mod_op->mod_id);
    if(existing_module != NULL) {
        uint8_t ver = sos_read_header_byte(
            existing_module->header, offsetof(mod_header_t, version));
        if (ver < mod_op->version) {
            sys_unload_module(existing_module->pid,
                sos_read_header_byte(existing_module->header,
                    offsetof(mod_header_t, version)));
        } else {
            return SOS_OK;
        }
    }
}
ret = fetcher_request(sys_DFT_LOADER_PID, mod_op->mod_id,
    mod_op->version, entohs(mod_op->size), msg->saddr);
s->pend = mod_op;
sys_led(LED_RED_TOGGLE);
return SOS_OK;
}
return SOS_OK;
```
Memory Leak in Module Loader: Discussion

Warning: Allocated data from instruction #line 125 "loader.c"
mod_op = (sos_module_op_t *) sys_msg_take_data(msg);
is not stored

- Memory is allocated
- Program checks to see if node can accept and should accept an incoming module
- Two paths corresponding fail to store, free, or release the data
Memory Leak in Module Loader (2)

- Two months later the program is restructured
- Old memory leaks remain and new leaks appear

```c
sos_module_op_t *mod_op;
if (msg->saddr == sys_id() || s->pend) {
    return SOS_OK;
}
mod_op = (sos_module_op_t*) sys_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
switch(mod_op->op){
    case MODULE_OP_INSMOD:
        return module_op_insmod(s,msg,mod_op);
    case MODULE_OP_RMMOD:
        return module_op_rmmod(s,msg,mod_op);
}
return SOS_OK;
```
Memory Leak in Module Loader (3)

```c
sos_module_op_t *mod_op;
if (msg->saddr == sys_id() || s->pend) {
    return SOS_OK;
}
mod_op = (sos_module_op_t*) sys_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
switch(mod_op->op){
    case MODULE_OP_INSMOD:
        return module_op_insmod(s,msg,mod_op);
    case MODULE_OP_RMMOD:
        return module_op_rmmod(s,msg,mod_op);
}
sys_free(mod_op);
return SOS_OK;
```

- Debugging session a few days later finds and fixes all memory leaks
Kernel Verification Results

<table>
<thead>
<tr>
<th>Directed memory leaks</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual dangling pointer errors</td>
<td>1</td>
</tr>
<tr>
<td>Missing annotations</td>
<td>10</td>
</tr>
<tr>
<td>Free within a loop</td>
<td>5</td>
</tr>
<tr>
<td>False positives</td>
<td>26</td>
</tr>
</tbody>
</table>

- Kernel verification examined 37 source files (about 15000 SLOC) of which 16 generated warnings
- Helped removed bugs in the current version of SOS
Dereference of Dangling Pointer

- Module dereferences the fst pointer after releasing it
- Is this a bug?
  - Violates the model
  - However it works with today's implementation of SOS...
  - This may soon change

```c
if( sys_post(fst->requester,
             KER_FETCHER_PID,
             MSG_FETCHER_DONE,
             sizeof(fetcher_state_t),
             fst,
             SOS_MSG_RELEASE) != SOS_OK ) {
    ...
    return;
}
cam = sys_cam_lookup( fst->map.key );
...```
Outline

- Overview of resources in sensor networks
- Model for proper resource usage
- A tool to automate verification
- Evaluation of verification
- Closing Remarks
Related Work

- Static checking in sensor networks
  - Concurrency in TinyOS
  - Typed components in galsC
- General reliability in sensor networks
  - Components, composition, aggregation...
- Formal programing languages
  - Ownership types
  - Confinement
  - Linear pointers
Future Work

- Verification with application model
  - Current checker does not “understand” the application
  - Find low overhead technique to capture this higher level understanding

- Formal type systems

- Link time and run time checking
Questions?
The End
Really. Nothing to see here.
“Property Number Three”

□(access(x) → ◻(alias(x, y) ∧ [alloc y ∨ get y]))

- Access to a “handle” should be preceded by an allocation or retrieval from a store
- What would a “get” operator look like in C code?
- How about a “set” operator to save data into a store?
- Should “get” be destructive? Is “get” destructive?
- What is the precise meaning of “access”? 