VM*: Synthesizing Scalable Runtime Environments for Sensor Networks

Joel Koshy, Raju Pandey
Department of Computer Science
University of California, Davis

ACM SenSys 2005

Presented by Yu-Han Li
Outline

- Introduction
- VM* framework
  - Programming model
  - Composition of system software
  - App. and system software evolution
- Implementation of core components
- Evaluation
- Conclusion
Introduction

- Sensors are being deployed at massive scales, containing a range of platforms
  - virtual machines as a means to address these challenges

- In order to satisfy the resource limitations
  - export only a minimal set of services to App. Programmer
Introduction

VM* - a framework for building resource-efficient virtual machine

- building and maintaining runtime environments for WSN App.
- Key insight that VM only needs to provide services needed by App. running on device
VM* framework

```java
char eList, eVector;
byte sHandle;
eList = Select.setEventId(eList, Events.PHOTOSENSOR | Events.RADIO_RECV | Events.TIMEOUT);
sHandle = Select.requestSelectHandle();
...
while (true) {
    ...
    eVector = Select.select(sHandle, eList); // blocking call
    if (Select.eventOccurred(eVector, Events.PHOTOSENSOR)) {
        handlePhotoSensor();
    }
    if (Select.eventOccurred(eVector, Events.RADIO_RECV)) {
        handlePacket();
    }
    ...
}
```
VM* framework

- Action listener model
  - The native event handlers invoke callbacks to perform application specific handling
  - Ex: Application specific event handling classes are extended form default handlers in library classes

- Centralized control point for dispatching handler
  - select call

- Decentralized control through implicit callbacks
  - action listener
VM* framework

- Composition of system software
  - Important steps in generating a runtime environment
    - Fine-grained specification of the system software
      - components
    - Using the spec. to generate a system software stack
  - A component language called BOTS
    - In which components include the notion of “attributes”.
Figure 3: Synthesizing a VM for a device and application.
VM* framework

- App. and system software evolution
  - With synthesized system software
    - updated App. depend on additional system component

- Incremental binary update solution
  - Diff is computed with the original program image
  - Diff algorithm only tracks structural changes
    - Code shift
VM* framework

- Our approach
  - Delta sizes can be reduced if code shifts are reduced
    - code shift - reflect structure changes and not necessarily changes in App. Semantics
  - When laying out code in memory, functions are provided with a small amount of slop space.
Implementation of core components

- Application preparation
  - Java class contains redundant constant pool (CP) entries

- Our compact strategy
  - eliminate redundant string info. representing symbolic reference from the CP
  - remove class attributes and meta-info. used by debuggers
Implementation of core components

- Interpreter implementation
- Use a bytecode table containing address of bytecode implementations

```c
/* NON-THREADED DISPATCH */
while ( 1 )
{
    opcode = fetch_1_byte();
    i = opcode_table[opcode];
    i();
}
```
Implementation of core components

- Threaded dispatch
  - incorporate the dispatch operation at the tail of every bytecode implementation

```c
/* QUASI-THREADED DISPATCH */
static void *op_labels = {&op1, &op2...};
```

<table>
<thead>
<tr>
<th></th>
<th>Non-threaded</th>
<th>Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROMized</td>
<td>SRAM</td>
</tr>
<tr>
<td>Fetch</td>
<td>27</td>
<td>13</td>
</tr>
<tr>
<td>Decode/Start</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Loop</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: Interpreter overheads in clock cycles.
Implementation of core components

- Bytecode implementation
  - Java bytecode of iinc (increment local variable by constant)

```c
#ifdef QUASI_THREADED
   { 
   instruction_iinc:
#else
   void instruction_iinc()
#endif
{
   ul local_num;
   s4 val;

   local_num = fetch_1byte();
   val = (s4)stack_frame_getlocal( local_num );
   /* sign extend the const to a byte */
   val += ((s1)(fetch_1byte()));

   stack_frame_setlocal( local_num, val );
}
#ifdef QUASI_THREADED
   opcode = fetch_1byte();
   goto *opcode_labels[opcode];
#endif
```

Figure 5: Bytecode implementation.
Implementation of core components

Native interface

Require access to native functions to perform operations depend on underlying OS services

Native method Imp. can safely exchange data between VM and native data spaces

Address of native methods are read from the VM binary and patched into the method headers in App. Classes.

```java
// Java class
package senses.platform.mica2.net;

public class CC1000 {
    static byte state;
    static {
        CC1000.init();
    }
    private static native void init();
    public static native byte sendRadioMsg( byte []data, char sz );
    public static native byte asendRadioMsg( byte []data, char sz );
    public static native byte sendRadioMsgUntilTimeout
        ( byte []data, char sz, long ms );
    public static native char recvRadioMsg( byte []data );
    public static native char arecvRadioMsg( byte []data );
    public static native char recvRadioMsgUntilTimeout
        ( byte []data, long ms );
}

/* Native (C) implementation */
RETURNTYPE_BYTE Java_CC1000_sendRadioMsg() {
    JReference array_ref;
    ul result, n;
    ul *buf;

    array_ref = GetReferenceParameter( 1 );
    n = (ul) GetCharParameter( 0 );
    buf = (ul *) GetArrayContents( array_ref );
    result = send_radio_msg( buf, n );
    ReturnByte( result );
}
```

Figure 6: CC1000 radio native interface.
Implementation of core components

- Operating system support
  - OS* is a scalable component-based O.S. that provides the concurrency framework.

Figure 7: VM* concurrency example: running three threads with outstanding events.
Evaluation

Platform
- Threaded interpreter with ROMized classes
- Applications in VM*, TinyOS, Mate
- Performance studies were down through AVRORA
Evaluation

- Interpreter performance (run lasting 480s)

<table>
<thead>
<tr>
<th></th>
<th>Non-threaded ROMized</th>
<th>Threaded ROMized</th>
<th>%</th>
<th>SRAM</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>69994.55</td>
<td>81019.32</td>
<td>15.75</td>
<td>114306.32</td>
<td>63.31</td>
</tr>
<tr>
<td>II</td>
<td>38712.34</td>
<td>47585.66</td>
<td>22.92</td>
<td>56992.42</td>
<td>47.22</td>
</tr>
<tr>
<td>III</td>
<td>27346.31</td>
<td>30900.75</td>
<td>13.00</td>
<td>31162.05</td>
<td>13.95</td>
</tr>
</tbody>
</table>

Table 2: Interpreter instruction issue rate (instructions per second).
Evaluation

- Energy overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>VM</th>
<th>TinyOS</th>
<th>Maté</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL</td>
<td>0.23</td>
<td>0.06</td>
<td>6.14</td>
</tr>
<tr>
<td>CTR</td>
<td>6.48</td>
<td>6.13</td>
<td>6.43</td>
</tr>
<tr>
<td>RTL</td>
<td>6.04</td>
<td>5.65</td>
<td>6.63</td>
</tr>
<tr>
<td>STR</td>
<td>5.92</td>
<td>5.78</td>
<td>6.62</td>
</tr>
<tr>
<td>Surge</td>
<td>6.09</td>
<td>5.90</td>
<td>6.20</td>
</tr>
</tbody>
</table>

Table 4: CPU activity as percentage of total time in active mode.
Evaluation

- Memory footprint

![Graph showing original and compacted class sizes](image)

**Figure 9**: Original and compacted class sizes.
Evaluation

Figure 10: Memory footprints of application and system software.
Evaluation

- Incremental linking

<table>
<thead>
<tr>
<th>Application pair</th>
<th>copy</th>
<th>run</th>
<th>add</th>
<th>Data (bytes)</th>
<th>% Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure diff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTL → CTR</td>
<td>1125</td>
<td>6</td>
<td>671</td>
<td>13233</td>
<td>45.26</td>
</tr>
<tr>
<td>CTR → Surge</td>
<td>1426</td>
<td>5</td>
<td>812</td>
<td>15280</td>
<td>43.87</td>
</tr>
<tr>
<td></td>
<td>Incremental linking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTL → CTR</td>
<td>884</td>
<td>13</td>
<td>495</td>
<td>9070</td>
<td>31.02</td>
</tr>
<tr>
<td>CTR → Surge</td>
<td>734</td>
<td>15</td>
<td>428</td>
<td>8161</td>
<td>23.43</td>
</tr>
</tbody>
</table>

Table 7: Delta footprint with pure diff and incremental linking.
Conclusion

- Through fine-grained software synthesis of a virtualized architecture,
  - An optimized execution engine and
  - A flexible model for extensibility,

the framework archives a balance between a number of competing concerns.