Hardware/Software Partitioning of Operating Systems: a Behavioral Synthesis Approach

Outline

- Motivations
- State of the art
- Traditional system overview
- Our solution:
  - Overview
  - Communication APIs
  - Context switching
  - Hardware support
  - HW-RTOS full architecture summary
- Case study and Results
- Conclusions
Motivations

- History of operating systems:
  - Initially created for supporting and scheduling the tasks that run on a CPU
  - Now they are a very complex infrastructures
  - Modern operating systems are designed to schedule and support any conceivable combination of applications

- This strategy makes sense for desktop systems, workstations and mainframes, but...

- It is not adequate for embedded systems

Embedded Operating Systems

- In embedded systems:
  - memory size is a constraint
  - response time is very often critical (real time embedded systems)

- In an SoC, each processor should perform at least two tasks:
  - the task for which it is designed
  - the task that provides communication with the rest of the system

- Any such processor that performs more than one task needs some kind of operating system to:
  - schedule the tasks
  - allocate resources
  - prevent deadlocks
Why hardware implementation of operating systems?

- Modern behavioral hardware synthesis tools allow to extract from the original Operating System kernel significant portions of its functionality and re-implement them in hardware without the need to perform major code modifications.
- How to achieve better performance:
  - select functionalities of a traditional operating system described in C language
  - synthesize them using a behavioral synthesis tool and integrate the new HW-RTOS with any bus interconnect

State of the art 1/6

- The idea of a hardware operating system has been addressed in some previous work.
- In all of these proposals, the main idea is to exploit the hardware acceleration by moving into hardware the functionalities that consume more CPU power.
State of the art 2/6

- **Silicon OS:**
  - it is a full-fledged operating system, in which the majority of the ITRON functionality is implemented on a coprocessor (*Silicon TRON*)
  - hardware support is provided for event flags, semaphores, timers, tasks, scheduler, control and interrupt management
  - memory and time management, translation of system requests and context switching are still implemented in software
  - the resulting software kernel is one third the size of the original software one

State of the art 3/6

- **FASTCHART:**
  - it is a real time kernel fully implemented in hardware
  - key features are: priority scheduling, synchronization primitives and interrupt handling
  - It has two components that run concurrently: CPU and Real Time Unit (RTU)
  - the CPU is designed to execute a context switch in one clock cycle
  - the RTU is implemented on an ASIC and can be interfaced with different system buses
  - the RTU has also been commercialized (*Sierra kernel*)
State of the art 4/6

- The δ Soc Codesign Framework:
  - it is built around the Atlanta kernel and allows a more fine-grained partitioning with respect to Silicon OS
  - it provides key RTOS features including multitasking capabilities, event-driven and priority-based preemptive scheduling, intertask communication and synchronization
  - the framework is designed to provide automatic configurability to support user-directed hardware/software partitioning of the Atlanta kernel

State of the art 5/6

- HOPES:
  - it is a RTOS-like system which allows run time partitioning and allocation of reconfigurable FPGAs
  - it supports both preemptive and non-preemptive scheduling methods

- System Weaver:
  - it is a hardware core that provides software designers a common task management and communication abstraction
  - it supports all popular methodologies (mutexes, semaphores, monitors and message passing) for seamless integration within existing applications and operating systems
State of the art 6/6

- Despite this amount of previous work and initial industrial attempts, at present, a commercial operating system does not take advantage of hardware to implement any of its functionalities.

- In “V. J. Mooney and D. M. Blough, A hardware-software real-time operating system framework for socs.” it has been pointed out that probably the reason is because processors and hardware accelerators have historically resided on separate chips.

- Partitioning the functionality of the RTOS between hardware and software was often impractical: the chip-to-chip communication would have overshadowed the amount of speed-up provided by hardware.

- But the situation is changing rapidly:
  - the advent of SoCs has replaced slow chip-to-chip communication with faster on-chip communication.
  - after the partitioning of the operating system hardware and software functionality can reside on the same chip.

A generic system

- A system can be described as a constellation of concurrent, interacting subsystems or tasks.

- A task consists of computation and communication nodes.

- Tasks can communicate using different communication styles:
  - message passing, using the concept of ports two APIs are provided:
    - Port Send
    - Port Receive (blocking and non-blocking)
  - shared memory.
Traditional approach OS POSIX-compliant

- Task1, Task2 and OS are implemented in software
- Communication nodes leverage the POSIX layer provided by the traditional OS
- Scheduler is also software

Traditional OS architecture

- The entire OS (Scheduler, Data Handling and Context Switch) is software running on the CPU
- Software tasks ($T_{sn}$) run on the operating system
- Hardware tasks ($Th_n$) are connected through the system bus
Proposed HW-RTOS Overview 1/2

- The POSIX support is replaced with dedicated data handling mechanisms
- The scheduler is also replaced
- In the original tasks:
  - the communication nodes need to be adapted in order to communicate with the HW-RTOS
  - the computation nodes remain unchanged
- Context switching is still performed by the embedded processor

Proposed HW-RTOS Overview 1/2

- Scheduler and data handling are hardware blocks
- Data and event buffers are used for handling the communication
- Communication nodes are adapted to be connected to the hardware data handling
Communication APIs

- The original communication APIs are automatically expanded by our interface synthesis tool without requiring any user interventions.
- The user can continue to use the same POSIX-based API without having to know about the presence of the HW-RTOS in the original implementation.
- The hardware part of the OS is automatically tailored by using the specification of the tasks.
- To handle the communication between OS, hardware and software tasks, a pool of port-event handlers is automatically generated to connect the HW-RTOS to the processor memory.

<table>
<thead>
<tr>
<th>API Primitive</th>
<th>eCos</th>
<th>HW-RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>port_receive (port, mode)</td>
<td><code>pthread_mutex_lock(&amp;p-&gt;mutex);</code></td>
<td><code>if (mode == NBLK) {</code></td>
</tr>
<tr>
<td></td>
<td><code>if (mode == BLK &amp; &amp; p-&gt;p-&gt;flag &amp; &amp; p-&gt;mask) {</code></td>
<td><code>v = port_receive_buffer[port];</code></td>
</tr>
<tr>
<td></td>
<td><code>v = p-&gt;p-&gt;value;</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td></td>
<td><code>p-&gt;p-&gt;flag ^= p-&gt;mask;</code></td>
<td><code>else {</code></td>
</tr>
<tr>
<td></td>
<td><code>if (!p-&gt;p-&gt;flag) {</code></td>
<td><code>SchedYield(port);</code></td>
</tr>
<tr>
<td></td>
<td><code>pthread_cond_broadcast(&amp;p-&gt;p-&gt;writer);</code></td>
<td><code>v = port_receive_buffer[port];</code></td>
</tr>
<tr>
<td></td>
<td><code>pthread_mutex_unlock(&amp;p-&gt;p-&gt;mutex);</code></td>
<td><code>frozen_input_events[port] = 0x00;</code></td>
</tr>
<tr>
<td></td>
<td><code>return v;</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td>port_send (port, data)</td>
<td><code>pthread_mutex_lock(&amp;p-&gt;mutex);</code></td>
<td><code>port_send_buffer[port] = data;</code></td>
</tr>
<tr>
<td></td>
<td><code>p-&gt;flag = p-&gt;total_readers;</code></td>
<td><code>active_input_events[port] = 0x1;</code></td>
</tr>
<tr>
<td></td>
<td><code>p-&gt;value = data;</code></td>
<td></td>
</tr>
</tbody>
</table>
Context switching

- It occurs when a software task executes a blocking port_receive

- It is performed by a software routine that:
  - pushes CPU registers into the stack
  - reads the identifier of the next software task and sets the new PCB
  - restores the context of the next software task

- It is implemented inside an interrupt service routine with assembly code specific for the target processor

- The routine is triggered by the signal containing the id of the last executed task

- Only one interrupt line is needed to handle all software tasks

```c
void SwitchTask() {
    // Push the CPU registers into stack
    push r0
    push r1
    push r2
    // Load the identifier of the next software task into r1
    // Load the PCB of the next task into r2
    // Restore the context of the next software task
    // The routine is triggered by the signal containing the id of the last executed task
    // Only one interrupt line is needed to handle all software tasks
}
```
Hardware support

- It is described in algorithmic C and then synthesized with the Cyber behavioral synthesis tool.
- It has three ports:
  - INPUT: CallRTOS (the task id of the last executed task), waitPort (the port on which the task is blocked)
  - OUTPUT: nextSWTask (communicates the scheduler decision)
- It consists of two main parts:
  - Initialization: all tasks are executed once
  - Main loop:
    - Data handling phase
    - Scheduling phase

```
/* SHARED MEMORY */
shared mem(SIZE; portvectorStart[NUM_PORTS];
shared mem(SIZE; portvectorBuffer[NUM_PORTS];
shared mem(SIZE; portqueueStart[NUM_PORTS];
shared mem(SIZE; portqueueBuffer[NUM_PORTS]);
shared mem(SIZE; activeInputEvents[NUM_PORTS];

/* PORT */
for(int i = 0; i < NUM_PORTS; i++) {
  portvectorStart[i] = portvectorBuffer[i];
  portvectorBuffer[i] = portqueueStart[i];
  portqueueStart[i] = portqueueBuffer[i];
  portqueueBuffer[i] = NULL;
}

while(1) {
  if(nextTaskPtr == NULL) {
    nextTaskPtr =_hwRTOS();
    if(nextTaskPtr == NULL) {
      nextTaskPtr = HWRTOS();
    }
  }
  switch(nextTaskPtr) {
  case 0:
    break;
  case 1:
    portSend(start); // Send data
    break;
  case 2:
    break
  }
} /* MAIN LOOP */
```

Behavioral C description of HW-RTOS

```
# Behavioral C description of HW-RTOS

- It is described in algorithmic C and then synthesized with the Cyber behavioral synthesis tool.
- It has three ports:
  - INPUT: CallRTOS (the task id of the last executed task), waitPort (the port on which the task is blocked)
  - OUTPUT: nextSWTask (communicates the scheduler decision)
- It consists of two main parts:
  - Initialization: all tasks are executed once
  - Main loop:
    - Data handling phase
    - Scheduling phase

```
/* SHARED MEMORY */
shared mem(SIZE; portvectorStart[NUM_PORTS];
shared mem(SIZE; portvectorBuffer[NUM_PORTS];
shared mem(SIZE; portqueueStart[NUM_PORTS];
shared mem(SIZE; portqueueBuffer[NUM_PORTS]);
shared mem(SIZE; activeInputEvents[NUM_PORTS];

/* PORT */
for(int i = 0; i < NUM_PORTS; i++) {
  portvectorStart[i] = portvectorBuffer[i];
  portvectorBuffer[i] = portqueueStart[i];
  portqueueStart[i] = portqueueBuffer[i];
  portqueueBuffer[i] = NULL;
}

while(1) {
  if(nextTaskPtr == NULL) {
    nextTaskPtr = hwRTOS();
    if(nextTaskPtr == NULL) {
      nextTaskPtr = HWRTOS();
    }
  }
  switch(nextTaskPtr) {
  case 0:
    break;
  case 1:
    portSend(start); // Send data
    break;
  case 2:
    break
  }
} /* MAIN LOOP */
```
Data handling phase

- When a task writes, it uses the *port_send_buffer* and the associated *active_input_event* is set (*port_send API*)
- Data and event are then copied into the *port_receive_buffer* and *frozen_input_events*

```
waitPort
```

```
callRTOS
```

```
Data handling
```

```
HardwareScheduler
```

```
nextSWTask
```

Scheduling phase

- Round robin loop where tasks are organized in a wheel
- *callRTOS* is received and then the task pointer is incremented
- Based on the priority created, the first schedulable task is returned with the signal *nextSWTask*, using a hardware interrupt

```
callRTOS
```

```
Priority Encoder
```

```
N: number of SW tasks
```

```
HW-RTOS
```

```
Port_send_buffer
Active_input_events
Port_receive_buffer
Frozen_input_events
```
HW-RTOS Architecture Summary

- The OS is partitioned:
  - Scheduler and Data Handling are hardware blocks
  - Context Switch is still software running on the CPU
- CallRTOS and waitPort are signals directed from the sw to the hw part of the OS and are routed through the bus
- nextSWTask is connected to the hardware interrupt port of the CPU
- Buffers and events are handled in the shared memory

Main reasons for speedup

- Inter-tasks communication:
  - The hardware can update memory locations by direct memory access, while the traditional OS generates bus transactions
- Scheduling:
  - The scheduling algorithm is implemented in hardware
Case study

- Software for Image filtering
  - Index Control
  - Data Retrieve
  - Filter
- Operating systems used:
  - eCos (with POSIX support)
  - HW-RTOS
- Synthesized with Cyber: NEC behavioral synthesis tool
- Simulated in Classmate: NEC cycle accurate hardware/software co-simulator

Case study – results 1/3

- Comparison for completing the full image filter:
  - Total speedup: 25.4
  - Initialization phase: 257.6
  - First iteration: 17.86
  - Context switching: 10.6

<table>
<thead>
<tr>
<th></th>
<th>Initialization</th>
<th>Index Control</th>
<th>Context Switch</th>
<th>Data Retrieve</th>
<th>Context Switch</th>
<th>Multiplier</th>
<th>Data Retrieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>eCos</td>
<td>407,274</td>
<td>12,197</td>
<td>10,048</td>
<td>18,268</td>
<td>10,048</td>
<td>71,545</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1st iteration = 122,106 clock cycles</td>
</tr>
<tr>
<td>HW-RTOS</td>
<td>1,507</td>
<td>2,394</td>
<td>944</td>
<td>1,289</td>
<td>944</td>
<td>2,549</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1st iteration = 8,112 clock cycles</td>
</tr>
</tbody>
</table>
Case study – results 2/3

- *Data handling & scheduling* in the HW-RTOS can be performed in parallel

Case study – results 3/3

- Comparison between port operations in eCos and HW-RTOS
Area Synthesis Results

- Technology library used:
  - 0.15 µm standard cell
- Number of equivalent gates:
  - ~10K (9280)
- Area:
  - 104,765 µm²

Conclusions

- The role of software is becoming more and more important in SoC design
- SoC architectures can significantly benefit from automated techniques for shifting functionalities of the OS into hardware
- We have shown some examples in which a small hardware area (less then 10k gates) results in 15X speedup
Thank you for your attention