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

Sathish Chandra
NEC Laboratories America
Princeton, NJ, USA
sathish@nec-labs.com

Francesco Regazzoni
ALaRI – Unversity of Lugano
Lugano, CH
regazzoni@alari.ch

Marcello Lajolo
NEC Laboratories America
Princeton, NJ, USA
lajolo@nec-labs.com

www.alari.ch **www.nec-labs.com**

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

NEC

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

NEC

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

NEC

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

NEC

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

NEC

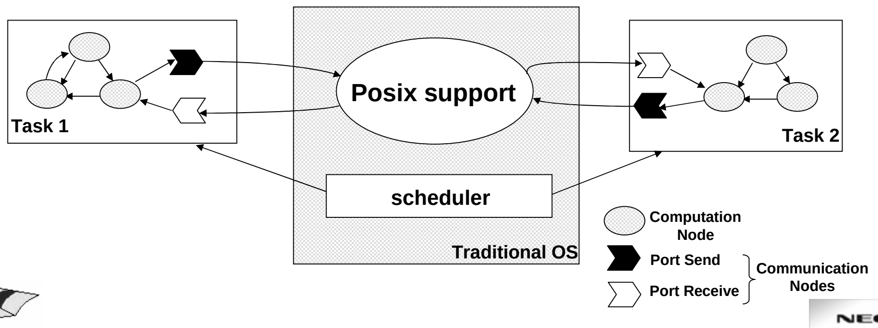
A generic system

- 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

NEC

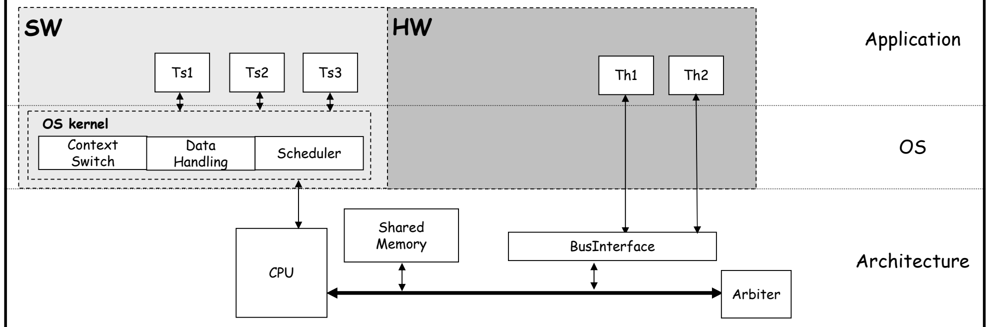
Traditional approach OS POSIX-compliant

- 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 (*Tsn*) run on the operating system
- Hardware tasks (*Thn*) are connected through the system bus

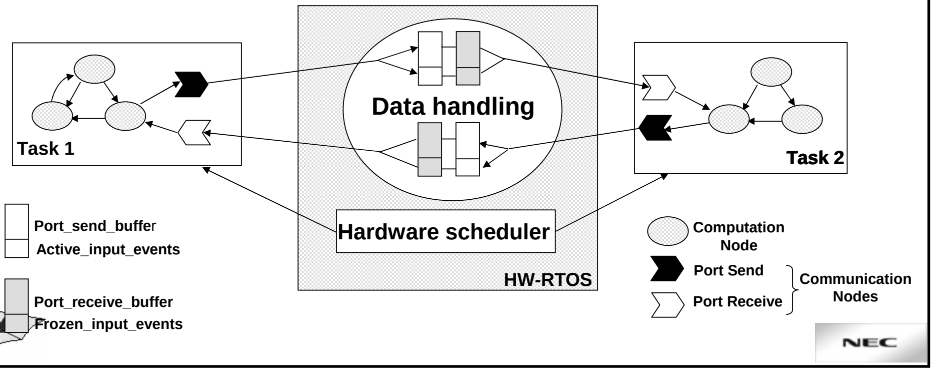


Proposed HW-RTOS Overview 1/2

- 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

NEC

Communication APIs comparison

- The interface is the same, to be compatible with POSIX layer
- Blocking ports have an associated event to signal new activity

API Primitive	Synthesized Code	
	eCos	**HW-RTOS**

port_receive
(port, mode)
	pthread_mutex_lock (&pr->p->mutex);
	if (mode == BLK && !(pr->p->flag && pr->mask)) {
	  pthread_cond_wait (&pr->p->reader, &pr->p->mutex);
	}
	v = pr->p->value;
	pr->p->flag ^= pr->mask;
	if (!pr->p->flag) {
	  pthread_cond_broadcast (&pr->p->writer);
	}
	pthread_mutex_unlock (&pr->p->mutex);
	return v;
		if (mode == NBLK) {
		  v = port_receive_buffer[port];
		}
		else {
		  SchedYield(port);
		  v = port_receive_buffer[port];
		  frozen_input_events[port] =0x00;
		}
		return v;

port_send
(port, data)
	pthread_mutex_lock (&p->mutex);
	p->flag = p->total_readers;
	p->value = data;
	pthread_cond_broadcast (&p->reader);
		port_send_buffer[port]=data;
		active_input_events[port] = 0x1;

NEC

Context switching

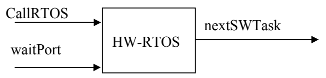
- It occurs when a software task executes a *blocking port_receive*
- It is performed by a software routine that:
  - pushes CPU registers into the stack
  - reads the identifier of the next software task and sets the new PCB
  - restores the context of the next software task
- It is implemented inside 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

Context Switch Routine

```
void SwitchContext() {
// Supervisor mode software interrupt function
// Push the CPU registers into Stack
asm("STMFD sp!,{r0-r3}"); // push r0-r3
asm("LDR r0,%0"::"m"(currPCB)"); // load pcb[0] into r0
asm("MRS r1, spsr"); // copy spsr into r1
asm("STMIA r0!, {r1}"); // store spsr into pcb[0]
asm("STMIA r0!, {lr}"); // store lr (return address) into pcb[1]
asm("ADD r0, r0, #16"); // location of r[4]
asm("STMIA r0!, {r4-r10}"); // store r4-r10 into pcb
asm("ADD r0, r0, #8"); // location of r[13]
asm("STMIA r0, {r13-r14}"); // store r13-r14 into pcb
asm("SUB r4, r0, #52"); // r4 = pcb_[2]
asm("LDMFD sp!, {r0-r3}"); // r0, r3 restored from stack
asm("STMIA r4, {r0-r3}"); // r0, r3 stored in pcb
asm("ADD r4, r4, #44"); // update r4 to point to pcb[13]
asm("LDMFD sp!, {r0}"); // pop from stack r11
asm("STMIA r4, {r0}"); // store r11 in pcb
asm("LDMFD sp!, {r0-r3}"); // pop from stack
asm("ADD r4, r4, #4"); // pcb[14]
asm("STMIA r4, {r3}"); // store r12 (ip) in pcb
// Get the NextSWTask from the HW-RTOS
currTASK= port_receive(&nextSWTask,NBLK ); // ...
currPCB= &PCB_[currTASK*PCB_OFFSET]; // ...
// Pop the CPU Registers from the stack
asm("LDR r0,%0"::"m"(currPCB)"); // load pcb_[0] into r0
asm("LDMIA r0!, {r1, r14}"); // copy PCB, ret address into R1, R14
asm("MSR spsr_fsxc, r1"); // copy the saved state register into usr spsr
asm("LDMIA r0, {r0-r14}"); // load from pcb r0-r14
asm("NOP"); // Debug
asm("MOVS pc, r14"); // update program counter
}
```

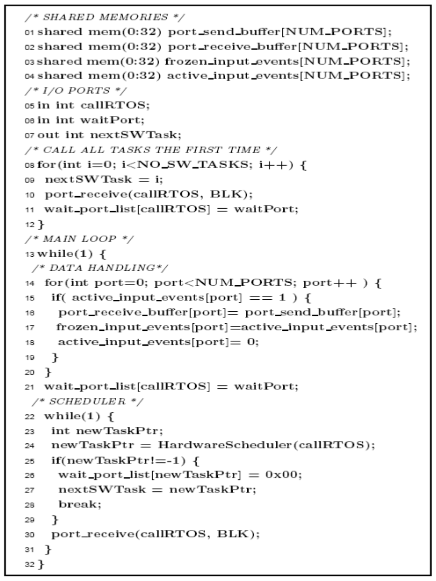
Hardware support

- 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

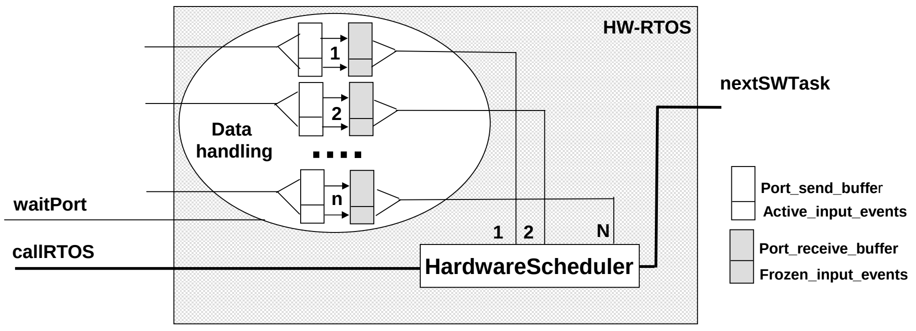
Behavioral C description of HW-RTOS



```
/* SHARED MEMORIES */
shared mem(0:32) port_send_buffer[NUM_PORTS];
shared mem(0:32) port_receive_buffer[NUM_PORTS];
shared mem(0:32) frozen_input_events[NUM_PORTS];
shared mem(0:32) active_input_events[NUM_PORTS];
/* I/O PORTS */
in int callRTOS;
in int waitPort;
out int nextSWTask;
/* CALL ALL TASKS THE FIRST TIME */
for(int i=0; i<NO_SW_TASKS; i++) {
 nextSWTask = i;
 port_receive(callRTOS, BLK);
 wait_port_list[callRTOS] = waitPort;
}
/* MAIN LOOP */
while(1) {
 /* DATA HANDLING*/
 for(int port=0; port<NUM_PORTS; port++ ) {
  if( active_input_events[port] == 1 ) {
   port_receive_buffer[port]= port_send_buffer[port];
   frozen_input_events[port]=active_input_events[port];
   active_input_events[port]= 0;
  }
 }
 wait_port_list[callRTOS] = waitPort;
 /* SCHEDULER */
 while(1) {
  int newTaskPtr;
  newTaskPtr = HardwareScheduler(callRTOS);
  if(newTaskPtr!=-1) {
   wait_port_list[newTaskPtr] = 0x00;
   nextSWTask = newTaskPtr;
   break;
  }
  port_receive(callRTOS, BLK);
 }
}
```

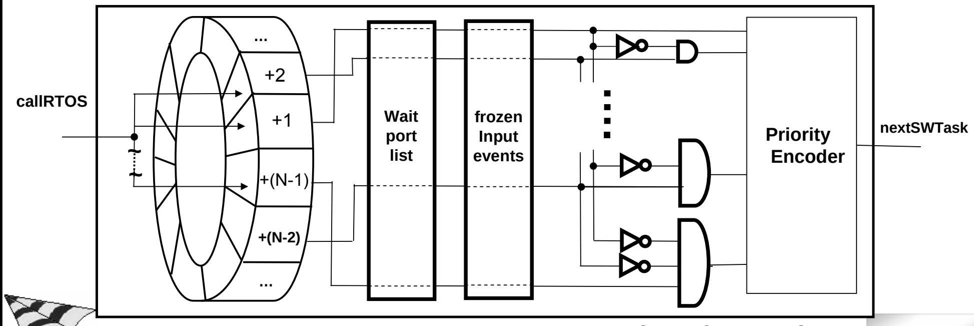
Data handling phase

- 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*



Scheduling phase

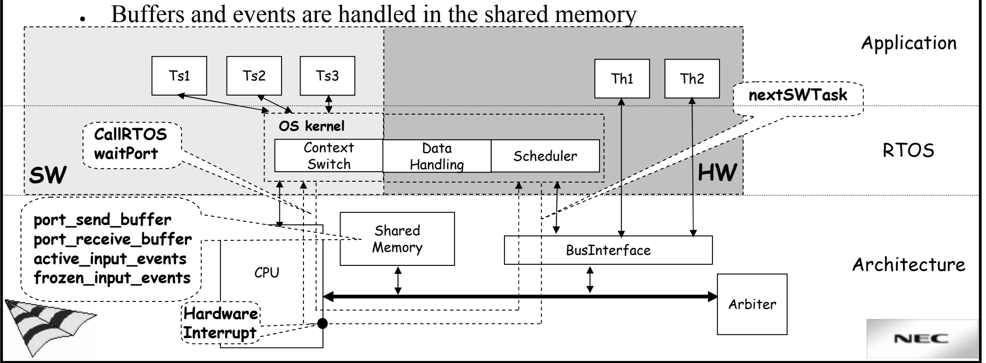
- Round robin loop where tasks are organized in a wheel
- *callRTOS* is received and then the task pointer is incremented
- Based on the priority created, the first schedulable task is returned with the signal *nextSWTask*, using a hardware interrupt



N: number of SW tasks

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

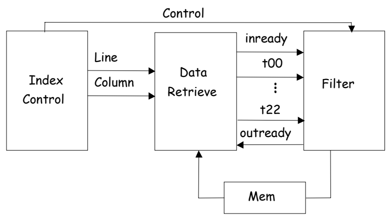


Main reasons for speedup

- 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

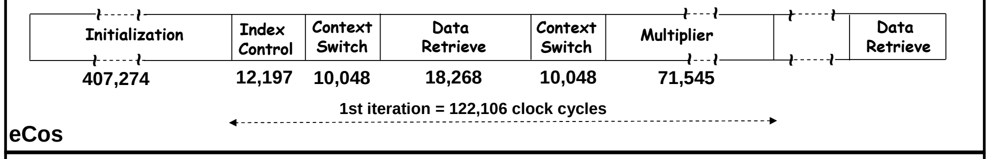
Case study

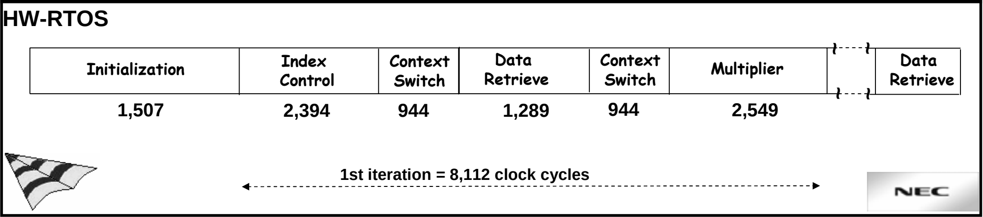
- Software for Image filtering
  - Index Control
  - Data Retrieve
  - Filter
- Operating systems used:
  - eCos (with POSIX support)
  - HW-RTOS
- Synthesized with Cyber: NEC behavioral synthesis tool
- Simulated in Classmate: NEC cycle accurate hardware/software co-simulator



Case study – results 1/3

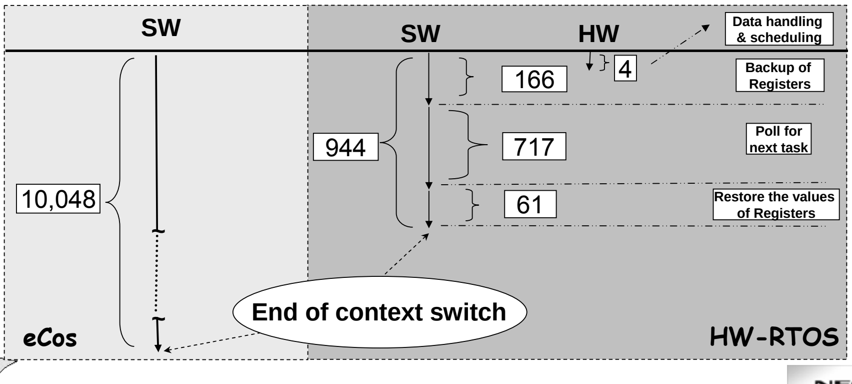
- Comparison for completing the full image filter:
  - Total speedup: 25.4
  - Initialization phase: 257.6
  - First iteration: 17.86
  - Context switching: 10.6





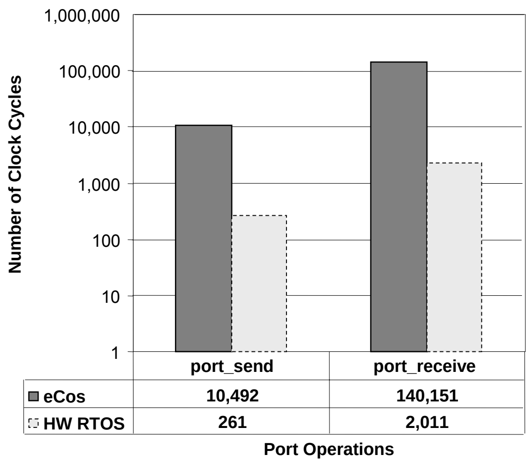
Case study – results 2/3

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

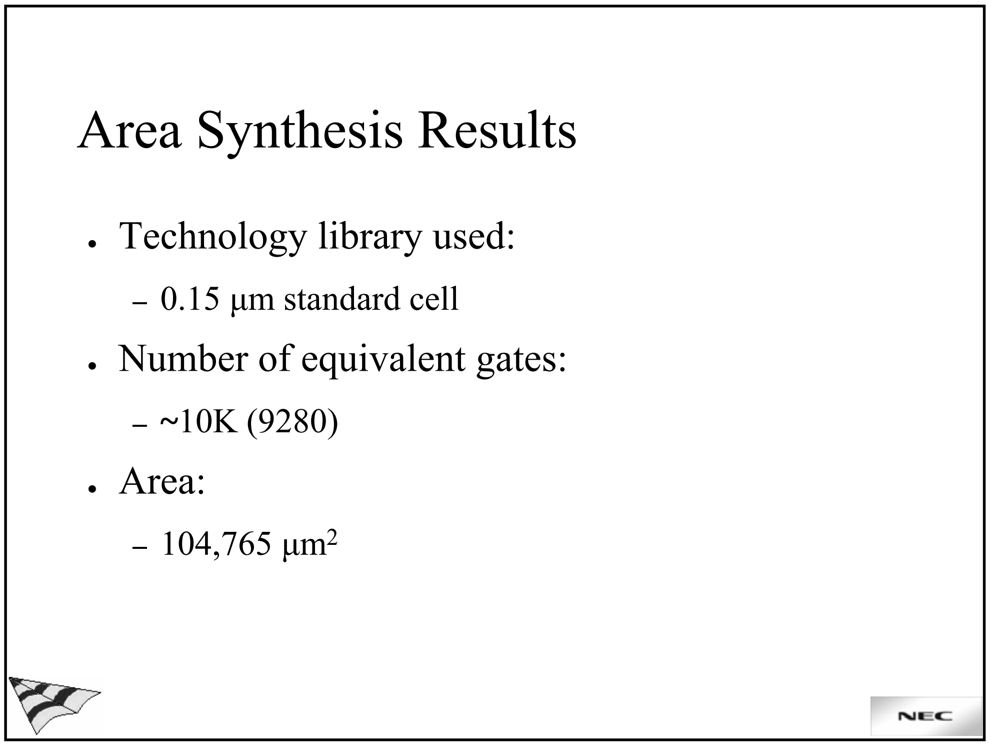


Case study – results 3/3

- Comparison between port operations in eCos and HW-RTOS

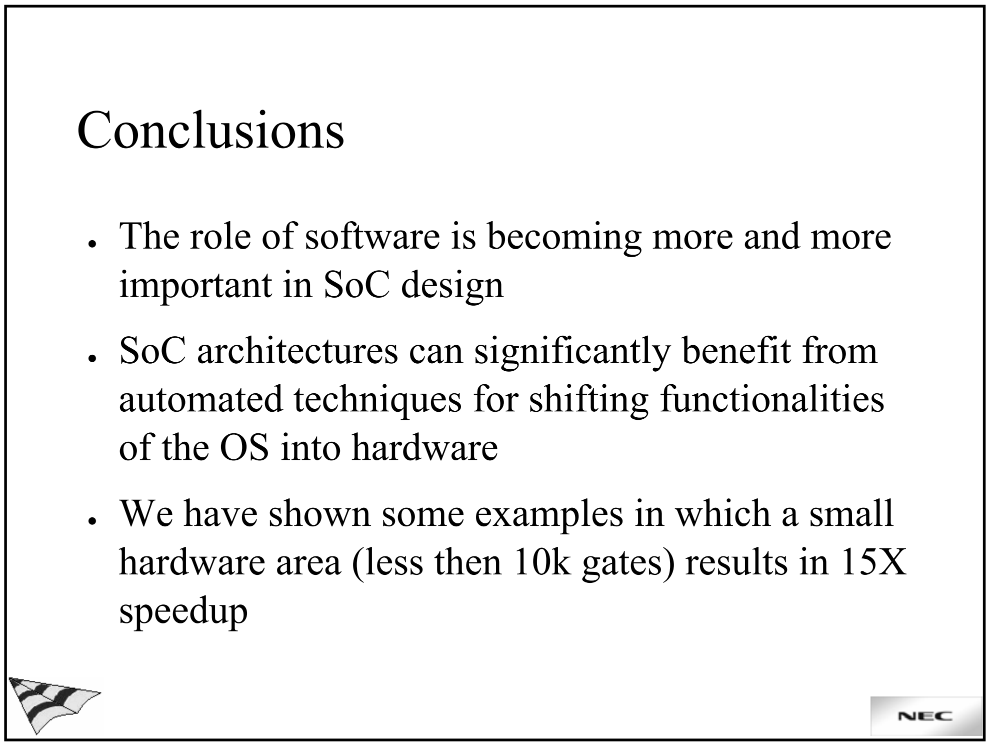


	port_send	port_receive
■ eCos	10,492	140,151
□ HW RTOS	261	2,011



Area Synthesis Results

- Technology library used:
  - 0.15 µm standard cell
- Number of equivalent gates:
  - ~10K (9280)
- Area:
  - 104,765 μm^2



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 for attention