

Porting Choices to ARM Architecture Based Platforms

Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

{fdavid,emchan,jcarlyle,rhc}@uiuc.edu

<http://choices.cs.uiuc.edu/>

Draft of 2007/01/12 15:12

Abstract

Choices, an object-oriented operating system, has been ported to the ARM architecture. The porting effort was based on the existing code-base for the SPARC processor based SPARCStation 1. Choices supports two ARM-based machine configurations: the Texas Instruments OMAP 1610 and the ARM Integrator as emulated by QEMU. Choices on ARM supports virtual memory, write-back data caching and interrupt management with support for serial I/O and timers. An environment consisting of X10-based remote power control devices, networked serial ports and comprehensive build scripts enable short development cycles with the OMAP hardware. The QEMU based virtual hardware presented powerful debug capabilities useful for developing an operating system for the ARM.

1 Introduction

Choices is an object-oriented operating system developed at the University of Illinois at Urbana-Champaign. The components of the system are encapsulated in classes and present a flexible design for management and extensibility. Choices originally supported National Semiconductor NS32332 based Encore Multimax machines, Sun SPARC machines [1] and Intel Pentium processor based machines [2].

The ARM architecture has become tremendously popular in the embedded devices market space, including the booming mobile device market. The modularity and extensibility inherent in Choices is a perfect match for the needs of a mobile device in a dynamic environment. This paper describes our efforts in porting Choices to two ARM-based platforms: the Texas Instruments OMAP1610 based H2 development boards [3] and the ARM Integrator CP platform [4] emulated by QEMU [5]. We first ported Choices to the OMAP hardware; experience from the OMAP port and the convenience of the QEMU emulator allowed the later Integrator port to be completed in a short time period.

We decided to use the SPARC port of Choices as the

initial code base for the ARM port. We chose to start with the SPARC port of Choices because it was the most used and tested port of Choices. All of the SPARC code was copied to an ARM directory and names were changed from SPARC to ARM. Machine dependent code was copied and renamed from SS1 (SPARCStation 1) to OMAP. All code that did not compile correctly was excluded from the build using preprocessor `#ifdef` statements and placing an `Assert(NOTREACHED)` statement which, when executed, causes Choices to stop and display an error message with the filename and line number where the `Assert` was encountered. The porting effort proceeded by repeatedly booting Choices and fixing excluded code whenever we encountered the corresponding `Assert` statement.

Apart from porting SPARC code to ARM, we also re-wrote parts of Choices for enhancements and clarity. Kernel debugging support was revamped and extended to support core functionality from the current version of the remote GDB protocol [6]. The original GDB functionality in Choices was confined to processor dependent code for SPARC. This was split into processor dependent and processor independent parts. Processor dependent GDB stub code was added for the ARM.

We also re-wrote the framework for handling processor interrupts. A machine and processor independent *InterruptManager* is used to handle and dispatch interrupts. A machine dependent class subclasses the *InterruptManager* and provides methods to interact with the hardware.

Pre-emptive context switching was also re-written for ARM. In the original design, when handling a time-slice interrupt, the interrupt thread would directly switch context to the new process. This interfered with proper functioning of the *InterruptManager*. So, the implementation was re-written so that the context switch only occurs when the interrupt thread returns back through the normal interrupt return mechanism.

During the porting process, we added support for the compressed ROM file system (cramfs) to Choices because all of the existing Choices file system drivers were

unusable. This is a read-only file system that was designed for simplicity and space efficiency. The cramfs file system is currently used as the root file system for both the Choices ARM ports. We discuss file system support in more detail in section 8.

One of the challenges encountered with porting Choices to the ARM platform was the lack of a suitable secondary storage device that we could use. The Choices source base includes hard disk drivers; however, our target ARM platforms do not include a hard disk drive. The OMAP and Integrator platforms include flash devices that could be used for secondary storage; however, Choices does not have a flash device driver. For the purposes of an initial port, we created a pseudo RAM disk device which stores all its data in volatile SDRAM. Choices expects a root file system to exist on this device and mounts this file system during the boot process. The file system image is pre-loaded into RAM by the boot-loader.

The remainder of this paper is organized as follows. A brief overview of the ARM architecture is presented in section 2. Some standards that are used for ARM software development are discussed in section 3. Section 4 is a high-level introduction to the Choices kernel architecture and organization. The computing environment used to develop the Choices ARM ports is described in section 5. Section 6 describes ARM processor dependent changes that were required. OMAP and Integrator machine dependent changes are documented in section 7.1. We discuss the file system implementations in Choices in section 8. Section 9 documents the development time-line for the porting effort. Section 10 has a short performance evaluation of the Choices ARM port. Section 11 summarizes our work and draws conclusions.

In this paper, words in italics, such as *InterruptManager* refer to Choices classes.

2 ARM Architecture Overview

ARM is a 32 bit RISC architecture. It supports a native full-fledged 32-bit instruction set. ARM also specifies two other instruction sets: a 16-bit compressed RISC set called Thumb, and an 8-bit instruction set for Java byte-codes called Jazelle. These are only available on selected versions of the ARM processor core. The ARM926EJ-S processor in the TI OMAP1610 H2 board and the ARM1026EJ-S processor in the Integrator conform to the ARMv5 architecture generation and support both these extended instruction sets. Newer processors use ARMv6 and ARMv7. Choices only has support for ARMv5 but does not use Thumb or Jazelle.

The ARM processors in both the OMAP and the Integrator support seven modes of operation. There are shown in table 1. All modes except ARM_USR have privileges to perform operations that control the MMU

and interrupts.

The ARM architecture has 37 registers as shown in table 2. 31 of these registers are general purpose registers including a program counter and 6 are status registers. Some of these registers are banked and are hidden except when executing in specific processor modes. These registers such as the stack pointer are automatically switched when entering a different processor mode. This design allows fast processing of interrupts as the handler code does not need to manually switch to a new stack.

An application normally has access to 16 general purpose registers (R0-R15) and one current program status register (CPSR). The following registers have special meaning: R15 is the program counter (PC), R14 is the link register (LR). By convention, the other general purpose registers are assigned meanings as well: R13 is the stack pointer (SP), R12 is the scratch register (IP), R11 is the frame pointer (FP) and R10 is the stack limit (which is currently unused). Registers R4-R9 are callee-preserved. Registers R0-R3 are used to pass arguments and return values when performing function calls. These register usage conventions are defined by an Application Binary Interface (ABI); the ABI is discussed in more detail in section 3.

A special set of banked registers, namely, saved program status registers (SPSR) are used to save a copy of CPSR when switching modes and are not accessible from the ARM_SYS or ARM_USR modes.

The ARM processor enters special interrupt handling modes when certain events occur. These events and modes are shown in table 3. When an interrupt is received, the PC register is set to either $0x00000000 + \text{offset}$ or $0xFFFF0000 + \text{offset}$ depending on whether the processor is configured to use low or high interrupt vectors.

The ARM processor supports address translation via an MMU (Memory Management Unit). The MMU uses a two-level page table to map virtual to physical addresses. Each first level page table entry can directly specify a mapping for a 1MB memory region. A first level entry may also indirect to a second level page table, which maps memory in increments of either 16 KB, 4 KB, or 1 KB. Both first level and second level entries include bits that determine access permissions based on processor mode. The entries also include bits that determine the caching policy used for the covered virtual memory region.

The ARMv5 architecture uses a split (Harvard) cache architecture with separate instruction and data caches. The cache can be disabled, configured for write-through or write-back for individual pages. An on-chip write buffer can be used to queue writes to memory. It also includes hardware extensions that support fast context switching without requiring a cache flush.

Table 1: ARM processor modes

Mode	Description
ARM_ABT	Abort - used for data and instruction fetch aborts
ARM_FIQ	FIQ - used for fast interrupts
ARM_IRQ	IRQ - used for normal interrupts
ARM_SVC	Supervisor - privileged mode for resets and software interrupts
ARM_SYS	System - privileged mode for operating system code
ARM_UND	Undefined instruction - entered when instruction decode fails
ARM_USR	User - unprivileged mode for user applications

Table 2: ARM Registers: Registers with an underscore in their name are banked registers

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15	R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
-	-	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Table 3: ARM Interrupt vectors and handling modes

Interrupt	Mode	Cause	Offset
Reset	ARM_SVC	Board is powered on or watchdog reset	0x0
Prefetch Abort	ARM_ABT	An instruction cannot be fetched from memory	0x4
Data Abort	ARM_ABT	A load or store failed	0x8
Illegal opcode	ARM_UND	An instruction could not be decoded	0xC
SWI	ARM_SVC	Software interrupt	0x10
IRQ	ARM_IRQ	An interrupt has occurred	0x14
FIQ	ARM_FIQ	A fast interrupt has occurred	0x18

3 ARM Software Standards

An Application Binary Interface (ABI) covers the specifications to which an executable must adhere in order to execute in a specific environment. An embedded ABI (EABI) is an ABI suited for embedded or free standing applications. The ARM EABI [7, 8] specifies conventions for executables such as file format and the format of exception handling tables. Executable files produced by different assemblers and compilers can be correctly linked together if they conform to the ARM EABI. The ARM EABI specifies a table-driven implementation of exception handling that has negligible impact on run-time performance. Codesourcery [9] has released a GNU g++ compiler tool-chain that produces EABI compliant binaries. The Choices ports to both the OMAP and Integrator are compiled using this tool-chain.

The ARM Procedure Call Standard [10] (APCS) is part of the ARM EABI and specifies low level subroutine call semantics that allows code generated from multiple compiler vendors and languages to work together. It describes a contract between the caller and the callee by defining conventions for register and stack usage during a subroutine call. APCS is supported by the GNU g++ EABI compiler using the compiler flag “-mapcs” and is used in the Choices ARM ports.

4 Choices Architecture Overview

The Choices kernel is implemented as a dynamic collection of interacting objects. System resources, policies and mechanisms are represented by objects organized in class hierarchies. The system architecture consists of a number of subsystem design frameworks [11] that implement generalized designs, constraints, and a skeletal structure for customizations. Key classes within the frameworks can be subclassed to achieve portability, customizations and optimizations without sacrificing performance [12]. The design frameworks are inherited and customized by each hardware specific implementation of the system providing a high degree of reuse and consistency between implementations.

A single instance of a machine dependent derived class of *Kernel* manages the operation of all subsystems in Choices. Each processor in the system is represented by a processor dependent instance of *CPU*. For example, for the SPARC port of Choices, there is one instance of *SSIKernel* and a per processor *SPARCCPU* object. Each process in Choices is represented by an object that implements the interface specified in the *Process* class.

Synchronization primitives are encapsulated in *Lock* and *Semaphore* objects. Timers are modeled in *Timer* and are managed by *TimerManager* objects. Scheduling policies are implemented as derived classes of *ProcessContainer* which exports `add()` and `remove()`

methods. A *MemoryObject* represents a region in memory. An *AddressTranslation* manages physical to virtual memory translations.

The boot process in Choices starts with assembly code in an architecture dependent “boot.s” file. Once some initial machine specific setup is completed, the assembly code transfers control to `KernelStart()` in “Start.cc”. This initializes the serial port and displays a message on the console before calling an architecture independent `Main()` function. This sets up an initial memory allocator, calls into machine dependent code to set up the console object, calls all static constructors, instantiates the machine specific *Kernel* object and calls the `initialize()` method on the newly created *Kernel*. The architecture independent `initialize()` sets up all kernel data structures and creates initial processes that complete all kernel initialization and present a console prompt. Once `Main()` returns, `CPU::processorStart()` is called which starts the process dispatcher (idle process).

Interrupts are first dispatched to an architecture independent class which manages all interrupt handling within Choices. The *InterruptManager* is an abstract superclass and delegates the following methods to subclasses:

1. `getInterruptNumber()` - Determines the interrupt source and returns a unique number corresponding to the source.
2. `ackInterrupt()` - Acknowledges receipt of the interrupt to interrupt management hardware.
3. `enableInterrupts()` - Enables interrupts on all peripherals.
4. `enableInterrupt()` - Enables a specific interrupt for a peripheral device.
5. `disableInterrupts()` - Disables all peripheral interrupts
6. `disableInterrupt()` - Disables a selected peripheral interrupt

InterruptManager implements a method called `setInterrupt()` which is used to register interrupt handlers. When an interrupt is received, architecture dependent assembly code does the initial context save, then the *InterruptManager* dispatches the interrupt by first reading the interrupt source using `getInterruptNumber()`. Once the interrupt source has been identified, the registered handler for the particular interrupt is invoked. Handlers are objects that inherit from abstract class *Interrupt* and implement the `basicRaise()` method. All interrupts that signal errors are converted to C++ language exceptions [13].

Some *Interrupt* objects use a semaphore to signal a waiting *InterruptProcess*. The process runs with interrupts disabled and is entrusted with the task of inter-

Architecture independent code	Kernel/ Memory/ Schedulers/ FileSystems/
Machine dependent code	MachineDependent/OMAP/ MachineDependent/Integrator/ MachineDependent/GenericARM/ MachineDependent/SS1/
Processor dependent code	ProcessorDependent/ARM/ ProcessorDependent/SPARC/
Interface/include files	Includes/ Includes/Memory/ Includes/MachineDependent/OMAP/ Includes/MachineDependent/Integrator/ Includes/MachineDependent/GenericARM/ Includes/ProcessorDependent/ARM/
Configure/build scripts	Configure/System/OMAP/ Configure/System/Integrator/ Configure/System/SS1/

Figure 1: Choices source code layout

acting with the peripheral and servicing the interrupt. This is the normal protocol for device drivers in Choices that deal with interrupts. The console driver, for example, spawns a console input process that waits on a semaphore for an interrupt. When the process is woken up, it reads a character from the serial port and enqueues it into the input buffer.

Some relevant portions of the code layout in the Choices source tree is shown in figure 1. The machine dependent code written for the ARM ports was initially in `MachineDependent/OMAP/` and `MachineDependent/Integrator/`. Some of this code was refactored and moved into `MachineDependent/GenericARM/`. This refactoring effort is described in more detail in section 7.3.

The build tools, scripts and “Makefiles” are all in a separate directory hierarchy called `Configure`. Compiling Choices is a two step process. First, a processor dependent general purpose library needs to be built. This is done by running “make” in `Configure/Libraries/GeneralPurpose/arm/`. Then, the kernel is built by running “make” in a machine specific directory in `Configure/System/`.

5 Development Environment

Most of our code development was carried out on Texas Instruments OMAP1610 H2 boards. We used

the JTAG [14] interface on these boards to write the U-Boot [15] bootloader to protected flash memory. U-Boot downloads Choices kernel images over the network from an NFS share and boots them. The serial port interface was used as a console for debugging and testing. Our development environment was set up so that the serial ports were accessible over the network through serial servers. We also used X10 devices [16] to power cycle the boards through network connections. This allowed several developers to work offsite. We used several virtual OMAP1610 platforms from Virtio [17] as debugging tools during initial development. Once the Integrator port was completed, the QEMU emulation environment also proved useful for debugging. QEMU supports debugging using the open remote GDB protocol. This was easier to use and more customizable than hardware-based solutions and the closed-source Virtio tool.

The Choices kernel build system for ARM currently requires a host machine running a UNIX-like operating system and ARM cross compiler that runs on the host. Currently, we are using the Codesourcery version of the GNU g++ cross compiler tool-chain running on a x86 Linux system. In the past, we have also built several tool-chains based on the standard GNU g++ published source code; however, the Codesourcery tool-chain is currently the preferred Choices kernel build tool.

Choices build scripts and the “Makefile”

were enhanced to support fast development cycles. The command “make”, when executed in `Configure/System/OMAP/` or `Configure/System/Integrator/` builds the Choices kernel. For the OMAP, “make run” automatically resets a pre-configured board using X10, uploads the kernel and boots Choices after redirecting the networked serial port output to the current console. On the Integrator, “make run” spawns QEMU with the arguments required to boot and run Choices. For both the OMAP and the Integrator, this helps to speed up development by eliminating any extra time required to manually reset, load or boot Choices.

The Subversion [18] source code version control system is used to manage the Choices source code. Bugzilla [19] is used to track bug reports and fixes.

6 Processor Dependent Code

The initial boot code and interrupt handlers for the ARM were adapted from the boot code of the U-Boot bootloader. This is written in assembly and is available in the “boot.s” file in `ProcessorDependent/ARM/`. This code sets up the stacks for all processor modes in the banked stack registers, initializes SDRAM and zeros the .bss section of the kernel that holds uninitialized data. Interrupt handling code is also part of the assembly code in this file. The interrupt handlers save execution context onto the stack and call into C++ code in “AsmBridge.cc” to complete interrupt processing.

The context of any running ARM *Process* is represented by an *ARMContext* object. This object also has `checkpoint()` and `restore()` methods that are used when switching contexts between two processes. *ARMSystemContext*, *ARMApplicationContext* are derivatives of *ARMContext* and support processes running with system mode and user mode privileges respectively. *ARMUninterruptableSystemContext* is a subclass of *ARMSystemContext* and is used for interrupt handler processes that need to run with interrupts disabled.

Every *Process* has a kernel stack that is allocated as when its associated *ARMContext* is created. When the processor receives an interrupt, the low level assembly code in “boot.s” switches from the corresponding processor interrupt mode to `ARM_SYS` and ensures that the process is running on its allocated kernel stack. The context of the running process is saved onto the kernel stack as an *ARMInterruptContext* object. A pointer to this object is passed to the interrupt handling routines. After interrupt processing, the saved context is restored on the processor by the low level code in “boot.s”. If pre-emptive context switching is enabled and if the interrupted process has exhausted its allocated time slice, the code restores the context of a newly selected process

instead.

When performing a context switch, the processor must save the current set of general purpose registers and the status register. It should then restore the saved registers of a different process. When restoring context, Choices switches in both the CPSR and PC at the same time. It does this by first switching to `ARM_SVC` and copying the new CPSR to the SPSR. A special flag in the instruction that loads the PC register causes the SPSR to be copied to CPSR when the instruction is executed.

The ARM processor core itself is modeled in the *ARMCPU* class. It encapsulates support for dispatching interrupts to the interrupt manager, enabling or disabling global interruptability flags for the processor and switching between different processor modes. The memory management unit is managed by the *ARMMMU* class. This includes support for turning the MMU on and off and switching between different sets of page tables. The page tables themselves are encapsulated in an *ARMTranslation* object. This is a derived class of *TwoLevelPageTable* which is an architecture independent class that models a two level page table scheme. In Choices, the first level table is referred to as a pointer table and the second level table is called a page table. The entries in the pointer tables and page tables are also encapsulated in objects of the types *ARMPointerTableEntry* and *ARMPageTableEntry*. These objects manage the mapping of a physical-to-virtual translation to actual values in RAM. They also export a standard set of page protection levels to the rest of Choices. Though the architecture supports a variety of sizes for pages, Choices only uses 4KB pages.

Choices uses the on-chip instruction and data caches to improve execution performance. The data cache is configured for write-back support. All page tables that hold translations for memory-mapped I/O regions are marked as non-cacheable and non-bufferable (writes are not buffered). The data cache is cleaned and flushed whenever a new *ARMTranslation* is activated on the MMU; for example, during a context switch. The hardware supported fast address space switching functionality has not yet been incorporated into Choices.

We leverage ARM support for high memory interrupt vectors in order to be able to detect null pointer dereferences in kernel code. Interrupt vectors are at the default address 0x0 at boot time. During the boot process, Choices remaps its interrupt vectors to 0xFFFF0000, which is the processor supported location for high exception vectors and enables this functionality in the processor. The first page with address 0x0 is then marked invalid in the page tables. This ensures that all null-pointer dereferences in the kernel are detected.

The interrupt handlers for peripheral devices are defined by machine dependent code. Interrupt handlers

for some processor exceptions are defined and assigned by processor dependent code. The *ARMDDataAccessInterrupt* handler checks for page faults and loads pages from disk if necessary. If an appropriate mapping is not available, the faulting process is deemed to have encountered an error. The *ARMUndefinedInstructionInterrupt* also signals an error. The *ARMInstructionAccessInterrupt* signals an error if the executed instruction is not a BKPT instruction used for debugging. All errors are converted to C++ exceptions and are thrown in the context of the process that caused the exception.

7 Machine Dependent Code

7.1 OMAP

The Texas Instruments OMAP1610 H2 development board uses an ARM926EJ-S core and provides an environment for developing software to run on GSM smart phones. It includes a debug daughter board which provides Ethernet, JTAG debugging support and debug LEDs. The main board includes the CPU, serial I/O, USB, keypad, LCD, camera port, compact flash slot and SIM card slot.

The relevant parts of the physical memory map of the OMAP1610 H2 board is shown in table 4. The Choices binary was linked at the start address 0x10000000 so that it can be placed at the start of volatile memory for booting. The composition of the Choices kernel image for the OMAP is shown in figure 2. The interrupt vectors are placed in the first part of the binary and are therefore at address 0x10000000 when Choices is loaded into

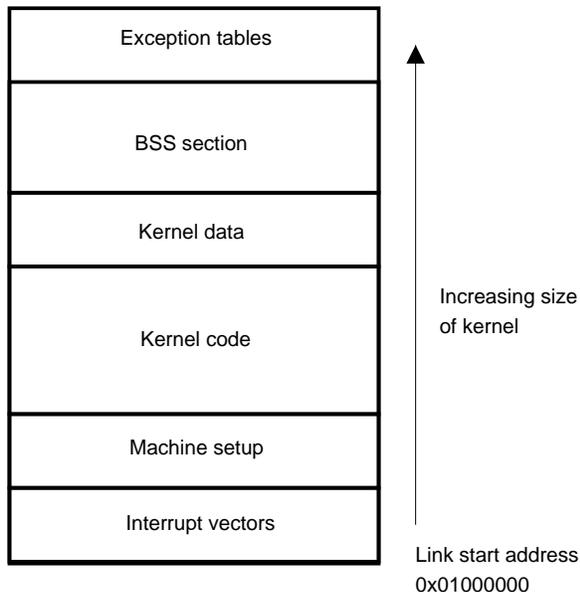


Figure 2: Choices OMAP kernel composition

SDRAM. The loading of Choices into SDRAM is accomplished by the U-Boot bootloader [15] which resides in write-protected flash memory at physical address 0x00000000. When Choices turns virtual memory on, it remaps one page at physical address 0x10000000 to 0xFFFF0000 and turns on high interrupt vectors support.

A RAM disk is loaded by the bootloader before the Choices kernel is booted. It holds a cramfs [20] compressed file system which includes some test user space programs and acts as the initial root file system when Choices boots. Choices' file system support is discussed in more detail in section 8.

The serial port on the OMAP H2 board is a NS16550 serial port. The *OMAPSerial* class encapsulates all interaction with the serial port. During the initial part of the boot process, serial input is received using polling. Once interrupts are turned on, input is interrupt driven.

The OMAP architecture includes a 2-level interrupt handler. Interrupt Handler Level 1 (IH1) manages 32 interrupts. Interrupt Handler Level 2 (IH2) manages 128 interrupts. IH2 is wired to interrupt 0 of IH1.

OMAPInterruptManager inherits from *InterruptManager* and implements all of its abstract methods. The constructor of this class configures IH1 and IH2 with parameters such as edge or level sensitivity and interrupt masks.

The OMAP1610 H2 board provides three 32-bit timers which can be programmed through memory mapped registers. Choices uses Timer 1 to drive the *FreeRunningTimer* which maintains wall clock time and Timer 2 to drive the *TimerManager* which is used for all system timers. The OMAP1610 also includes an external watchdog timer. A driver for this watchdog timer is available in *OMAPWatchdogTimer*. A memory-mapped status register logs the reason for a processor reset. It can be used to determine if the reset was due to a power-on event or due to a watchdog timeout. Choices prints out the reset reason during boot.

All machine dependent code is managed by *OMAPKernel* which is a subclass of the architecture independent *Kernel* class. The memory-mapped I/O regions are defined and added to the virtual memory map by routines in *OMAPKernel*.

7.2 Integrator

QEMU version 0.8 and above supports ARM system mode emulation. The emulated hardware is an ARM Integrator CP board with an ARM1026EJ-S processor core and various peripherals including timers, cascading interrupt controllers, serial ports and an Ethernet controller based on the SMC LAN91C111 chip. This emulated hardware is capable of running a full operating system such as Linux with networking capabilities.

Region	Start	End	Size (bytes)
Boot ROM	0000 0000	0000 FFFF	64K
SDRAM	1000 0000	11FF FFFF	32M
Devices	FFFB 0000	FFFE FFFF	256K
High Interrupt Vectors	FFFF 0000	FFFF 0FFF	4K

Table 4: OMAP1610 H2 Memory Layout

Region	Start	End	Size (bytes)
SDRAM	0000 0000	01FF FFFF	32M
Timers	1300 0000	1300 0FFF	4K
Primary Interrupt Controller	1400 0000	1400 0FFF	4K
Serial UART	1600 0000	1600 0FFF	4K
Secondary Interrupt Controller	CA00 0000	CA00 0FFF	4K
High Interrupt Vectors	FFFF 0000	FFFF 0FFF	4K

Table 5: Integrator Memory Layout

The memory map of the Integrator is shown in table 5. SDRAM memory on the Integrator begins at the address 0x00000000. QEMU pre-loads the kernel image at address 0x00010000 and the RAM disk at address 0x00800000 before it starts the emulation. The boot-loader is hard-coded into the QEMU emulation code at address 0x00000000 and sets up a couple of registers before jumping to the kernel at address 0x00010000. Choices uses high interrupt vectors for the Integrator as well.

The composition of the Choices kernel image for the Integrator is identical to that of the OMAP except for the absence of the RAM disk since it is pre-loaded by QEMU.

The ARM Integrator includes an ARM PL011 PrimeCell serial port. This is modeled in the class *IntegratorSerial*. Console operations are implemented in the class *IntegratorConsole*, which implements priority, exclusive access, string operations and interrupt-driven I/O.

The Integrator features a primary and a secondary interrupt controller. An interrupt source can be assigned to either an IRQ or FIQ on the processor. Each of these controllers manage 32 interrupts. The secondary controller is wired to interrupt 26 on the primary. Both interrupt controllers allow masking interrupts from various peripherals. The interrupt controllers are managed by the *IntegratorInterruptManager* class.

The Integrator board has three timers. Each timer may be programmed independently. At the time of writing, only the first timer is in active use as the OS timer.

All machine dependent code is managed by *IntegratorKernel* which is a subclass of the architecture independent *Kernel* class. The memory-mapped I/O regions are defined and added to the virtual memory map by rou-

tines in *IntegratorKernel*

7.3 GenericARM

A major refactoring of the OMAP and Integrator code was carried out in order to minimize code duplication across both these ports. We extracted machine dependent code that was identical or similar across both these ports and combined them into a GenericARM hierarchy. The current versions of OMAP and Integrator machine dependent classes inherit from classes in the GenericARM hierarchy

OMAPConsole and *IntegratorConsole* were replaced by *GenericARMConsole*. Some common code from *OMAPKernel* and *IntegratorKernel* was moved to *GenericARMKernel*. The machine interrupt manager classes were unique to the machines and could not be combined into a single GenericARM class. This was also the case for the serial port drivers. This refactoring effort resulted in a compact code-base that was more maintainable than the original. The current source tree only has machine specific device drivers in the OMAP and Integrator directories. All common machine dependent code has been moved into the GenericARM directory.

We also found it possible to move Choices support for GDB on ARM into the GenericARM area. The GDB debugger support that is built into Choices uses the ARM's BKPT instruction to trap at breakpoints. Execution of this instruction causes the processor to be interrupted with a Prefetch Abort interrupt. This case is distinguished from a genuine instruction fetch abort by reading the instruction and checking to see if it is a BKPT instruction. If kernel debugging is enabled, Choices passes control to an architecture independent GDB stub in *Gdb* which implements the GDB remote serial protocol [6].

All ARM specific code is in *GenericARMGdb* which inherits from *Gdb*.

8 File System Support

Earlier versions of Choices had support for numerous file systems including the BSD Fast File System (FFS) [21] and the MS-DOS FAT [22]; however, we found the existing Choices file system code to be in a state of disrepair. The bulk of the source code to support the FAT file system was unfortunately mistakenly deleted sometime in the past and does not exist in the Choices repository. The FFS code was found to have significant problems. Choices had originally been written for big endian systems, and most of the file system code was written to assume big endian systems. We investigated adding byte swapping code to the FFS implementation; however, it was determined that such refactoring would require a significant rewrite of the existing code.

Choices also included an implementation of Linux's Ext2 [23] file system that was developed as part of a class project. The Ext2 implementation also had several problems. It made assumptions about the layout of the disk that would not be true for the RAM disk used for the ARM platforms. Also, only read support had been completed during the class project. Ext2 write support was non-existent.

To simplify the porting effort, we decided that we would write an implementation of the cramfs file system for Choices. Cramfs was an easy target because it is a read only file system that stores compressed files and has a very simple disk layout. It was originally designed for small embedded devices and therefore fits in well with our target ARM platforms. Cramfs was easily incorporated into the existing Choices file system class hierarchy: we simply had to plug in the appropriate cramfs-specific functions. Even the most abstract of the Choices file system classes have a concept of an inode that can be used as an index to determine where a file's metadata and data reside. This concept does not work well with cramfs since its "inodes" are of variable length. We worked around this issue by using the offset of a file descriptor from the beginning of the disk as its pseudo-inode number. For decompressing the files, we used the open source zlib library.

Thanks to the addition of a usable file system, we were able to allow arbitrary user space binary applications to be loaded and executed. Choices supports applications binaries in the ELF [24] format. ELF is a common format for application binaries, and Choices applications can be compiled using off-the-shelf versions of the GCC tool-chain that create ARM ELF binaries. In order to support ELF binaries, a new ELF loader class had to be written.

One interesting example of code reuse in the Choices class hierarchy is the ELF file loader. As with the existing Choices COFF object file loader, the ELF file loader is implemented as a file system. That is, the ELF loader classes are derived from file system base classes. This organization means that it is possible to mount an ELF file just as one would mount a file system. Reusing the file system classes allowed us to avoid duplicating code for handling high level data structures and provided an existing interface that could be used to query ELF files.

Having a read-only file system suited our needs for sometime; however, a growing need for a read-write file system forced us to continue development on the incomplete Ext2 driver. We eventually fixed most of the problems with the Ext2 driver and added initial write support for creating directories and files. But, this code is still unstable and the cramfs file system is currently the recommended file system for the Choices ARM ports.

There are other areas of the Choices file system design that we are unhappy with. For instance, there is a top level FileSystemInterface class that is used to determine the file type. For instance, this class looks at a file and determines if it is an ELF file. Specific functions such as this should not be so tightly coupled to generic classes. We are currently investigating ways to break this coupling.

9 Development Timeline

The porting effort started in September 2004 and lasted until December 2005. Several portions of Choices were re-designed and re-written as well during this time period. The first six months were spent getting the source code to compile and getting virtual memory and context switching to work. After a short break over the summer, Interrupt management, timers, and interrupt driven serial I/O were implemented. Remote GDB support was also revamped and updated. The last couple of months were spent adding support for a RAM disk hosting a cramfs file system. Preemptive context switching was reworked and the Integrator port was completed by January 2006. All stability issues were resolved by early February and the codebase was used in several projects for the CS523 (Advanced Operating Systems) course in Spring 2006.

10 Performance

The porting effort did not really concentrate on performance optimization of various kernel operations. Unlike the Linux kernel, which has performance optimizations that take into account small details like cache line size, the Choices kernel is not yet fine-tuned for performance. Nevertheless, we report the results of a few interesting performance benchmarks in this section.

OS benchmarks normally include statistics such as the context switch time, system call performance, and

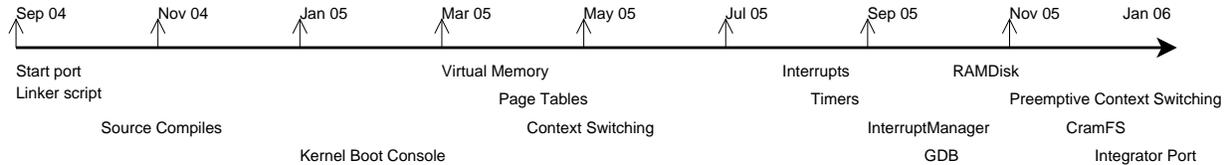


Figure 3: Choices ARM port timeline

various I/O performance numbers for filesystem access and networking. For example, the LMBench portable benchmark suite for Linux [25] measures all of these performance characteristics. For Choices, most performance measurement tools are built into the kernel and can be invoked by issuing special commands at the kernel prompt.

All performance numbers are measured on the OMAP H2 board with the ARM processor core clocked at 96MHz. The processor instruction and data caches are turned on with write-back caching enabled for data. Performance of I/O benchmarks are not yet available for Choices because the System Interface layer is being refactored and is currently not completely usable.

The context switching time for Choices was measured to be 12 microseconds. In comparison, LMBench on Linux running on the OMAP reports a context switch time of 180 microseconds. This comparison is not really fair because the Choices benchmark measures switching between two processes using a semaphore and LMBench measures switching using a pipe and encounters more overhead. These numbers, however, show that the Choices kernel performs quite well during context switching.

A simple null syscall on Linux as measured by LMBench takes 2.63 microseconds. In contrast, a system call on Choices currently takes 57 microseconds. This significantly worse performance can be attributed to the overhead caused by the object-oriented system call interface exported by Choices. We believe that further optimization will reduce this overhead and result in acceptable performance.

Choices also includes a UNIX compatibility library which can be used to port benchmarking tools such as LMBench to Choices, thus providing more accurate performance statistics comparisons. We are currently working on completing the System Interface layer and porting the existing UNIX compatibility library to the new System Interface layer.

11 Conclusions and Future Work

In this paper, we have described a high-level overview of the ARM architecture, the Choices kernel and details of our port of Choices to two ARM based machines. We have also documented our development environment

setup that supports fast development cycles. The complete porting effort lasted little over a year and also resulted in some Choices architecture redesign. For future work, several additional features like support for the ARM fast context switch extension, support for memory domains and support for multiple-granularity pages are planned. Some of these features should substantially improve kernel performance. Existing Choices code also requires some tuning and small refactorings to maximize performance.

The Choices port to Integrator was used quite extensively in other projects. Students who did not have access to real hardware could easily develop code using QEMU. We have also used QEMU as a platform for some fault-injection and virtualization experiments.

Choices on ARM is currently the most stable port of Choices and is being used as the base for several class projects and research into reliable operating systems and virtualization.

Acknowledgments

We would like to thank David Raila for discussions that led to some of the work described in this paper. We would also like to thank Motorola, for providing Jeff and Ellick with employment during the development of Choices. Part of this research was made possible by a grant from DoCoMo Labs USA and generous support from Texas Instruments and Virtio.

References

- [1] David Raila. The Choices Object-oriented Operating System on the Sparc Architecture. Technical report, The University of Illinois at Urbana-Champaign, Aug 1992.
- [2] Lup Lee. PC-Choices Object-oriented Operating System. Technical report, The University of Illinois at Urbana-Champaign, Aug 1992.
- [3] Texas Instruments OMAP Platform. <http://focus.ti.com/omap/docs/omaphomepage.tsp>.
- [4] ARM Integrator Family. <http://www.arm.com/miscPDFs/8877.pdf>.
- [5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

- [6] GDB Remote Serial Protocol. http://sources.redhat.com/gdb/current/onlinedocs/gdb_33.html.
- [7] Application Binary Interface for the ARM™ architecture. <http://www.arm.com/miscPDFs/8061.pdf>, March 2005.
- [8] Exception Handling ABI for the ARM™ architecture. <http://www.arm.com/miscPDFs/8034.pdf>.
- [9] Codesourcery. <http://www.codesourcery.com/>.
- [10] Procedure Call Standard for the ARM Architecture. <http://www.arm.com/miscPDFs/8031.pdf>, October 2005.
- [11] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices, Frameworks and Refinement*. In Luis-Felipe Cabrera and Vincent Russo, and Marc Shapiro, editor, *Object-Orientation in Operating Systems*, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society Press.
- [12] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: a Case Study. In *USENIX C++ Conference*, pages 103–114, San Francisco, CA, April 1990.
- [13] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. *Exception Handling in the Choices Operating System*. Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 2006.
- [14] IEEE 1149.1-1990. IEEE Standard Test Access Port and Boundary-Scan Architecture, 1990.
- [15] Das U-Boot - Universal Bootloader. <http://u-boot.sourceforge.net/>.
- [16] X10. <http://www.x10.com>.
- [17] Virtio. <http://www.virtio.com/>.
- [18] Subversion. <http://subversion.tigris.org/>.
- [19] Bugzilla. <http://www.bugzilla.org/>.
- [20] Cramfs. <http://en.wikipedia.org/wiki/Cramfs>.
- [21] Marshall K. McKusick, William N. Joy, Samuel J. Lefler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [22] Charlie Russel. NTFS vs. FAT: Which Is Right for You? http://www.microsoft.com/windowsxp/using/setup/expert/russel_october01.aspx, October 2001.
- [23] Ext2. <http://en.wikipedia.org/wiki/Ext2>.
- [24] Executable and Linkable Format. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
- [25] Larry W. McVoy and Carl Staelin. Imbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.