A Raspberry Pi Operating System for Exploring Advanced Memory System Concepts

Pascal Francis-Mezger
University of Maine
pascal.francismezger@maine.edu

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

ABSTRACT

Modern memory hierarchies are complex pieces of hardware, combining memory controllers, caches, and memory-management units in a desperate attempt to keep modern high-speed processors fed with data. Recent developments, including the introduction of non-volatile memory, further complicate the situation.

Operating systems are highly tuned to current memory systems; modifying them to take advantage of new developments is a difficult process. The low-level code involved is complex and hard to follow. This makes teaching students about modern memory systems a struggle, as wading through the complicated code in a full operating system like Linux can be frustrating.

To this end we develop vmwOS, a simple operating system designed for low-cost Raspberry Pi development boards. Although inexpensive, these widely used boards support most of the features of modern memory systems, including 64-bit addresses, multi-level caches, multi-core, and full ARMv8 processor and MMU support. vmwOS is simple enough that students can follow the code and make modifications for memory hierarchy exploration.

We describe a number of memory research topics that can be explored with this infrastructure, including a detailed examination of simulating non-volatile RAM support.

CCS CONCEPTS

• **Applied computing** → *Computer-assisted instruction*; • **Software and its engineering** → *Virtual memory*; • **Information systems** → Flash memory; • **Hardware** → Non-volatile memory;

KEYWORDS

Operating Systems, Memory Systems, Raspberry Pi

ACM Reference Format:
Pascal Francis-Mezger and Vincent M. Weaver. 2018. A Raspberry Pi Operating System for Exploring Advanced Memory System Concepts. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3240302.3240311

1 INTRODUCTION

Modern computer memory hierarchies are complex, combining various interacting levels of caches, memory controllers, and memory technologies in an effort to hide the wide gap between memory and processor access latencies. This long-standing latency disparity issue is often called the Memory Wall [43]. In addition to the caches involved, most modern systems have some sort of virtual memory infrastructure which adds its own complications and latencies. The introduction of new memory technologies, including non-volatile memory, is only going to make the situation worse.

It is difficult to write code that can obtain high performance in the face of such a complicated underlying architecture. User code often relies on the operating system to handle the setup, configuration, and manipulation of the memory hierarchy. The memory-related code in the operating system becomes just as complex and hard to understand as the underlying hardware. Current operating systems end up highly tuned to existing memory systems; modifying them to take advantage of recent developments in memory system design can be a difficult process.

All of this complexity makes it difficult to teach about modern memory systems and their interactions with system software. Wading through the relevant code in a full-featured operating system like Linux can be a frustrating exercise.

To make things easier for students we propose vmwOS, a simple operating system designed for low-cost Raspberry Pi development boards. Although inexpensive, these widely used boards support most of the features of modern memory systems, including 64-bit addresses, multi-level caches, multi-core, and full ARMv8 processor and MMU support. The operating system is simple enough that students can understand it, make modifications, and explore the modern memory landscape.

We describe the hardware found on Raspberry Pi boards, the features of the vmwOS, and then propose various memory system explorations that can be done. This includes investigating the benefits and tradeoffs of the various levels of caches, hands-on manipulation of the virtual memory infrastructure, and experimentation with non-volatile RAM concepts.

2 COMPLEXITY IN MEMORY EDUCATION

Memory systems are critically important to modern processor performance, however the topic is complex enough that it is difficult to cover in depth during a typical computer architecture course. The topics involved can fill their own 1000 page long textbook [19].

There are a few common ways of teaching memory hierarchies which we elaborate on below.

MEMSYS, October 1–4, 2018, Washington, DC, USA

ACM ISBN 978-1-4503-6475-1/18/10...$15.00
https://doi.org/10.1145/3240302.3240311

2.1 Simple Cache Examples

Often in a semester-long undergraduate computer organization or computer architecture class there is time to spend a few weeks on caches and virtual memory. A typical way to do this is to go over the concepts, then to go through a handful of contrived cache examples. These may not go through more than a few tens of instructions, and the caches involved are simplified hierarchies of the type last found in desktop machines in the 1990s.

While students usually can work out on paper the cache miss behavior of these simple caches, it often does not translate to a deep intuitive feel for the problems facing modern memory systems.

2.2 Hardware Measurements

Another way to investigate the memory system is to take measurements on actual hardware. Most modern systems support hardware performance counters, which can allow measurement of architectural events that occur during program execution. This includes things such as cycles, branch predictor misses, and retired instructions, but also memory system events such as cache hits and misses and TLB misses. More advanced processors can report things such as detailed DRAM statistics, cache coherency events, inter-processor and off-core memory transfers, and sampled cache latency values. These results can be gathered on Linux using tools such as perf [14].

While gathering these cache and memory statistics is straightforward, interpreting them can be difficult. Even deterministic events such as retired instructions can have cases of overcount and nondeterminism [42]. Non-deterministic events such as cache misses can have surprising results, especially when run on systems with other programs running. Advanced hardware features like hardware prefetching can cause surprisingly low cache miss rates that do not match the simple models typically taught in classes. Many modern chips have fairly comprehensive sets of errata involving the memory-related performance counters, again causing confusion when the results gathered do not match expectations. Even determining the expected, proper, results can be difficult, as full parameters for the cache hierarchy are proprietary information and are not generally available for most processors, especially high-end desktops and servers.

Due to these issues hardware performance counters can be a valuable tool for exploring memory hierarchies, but the results often lead to more questions, rather than answers.

2.3 FPGA Implementation

One way around the problem of not knowing the details of a commercial memory hierarchy is to build your own from scratch. These days it is impractical to fab your own chip in actual silicon, but it is possible to make a design using a field-programmable gate array (FPGA) [34]. FPGAs are essentially re-programmable hardware, and can be used to design full memory hierarchies.

One downside to using FPGAs in a class is that unless the students come in with previous FPGA knowledge, it can take a large amount of time to bring them up to speed on the various tools and techniques needed. A full memory hierarchy can be a complicated thing to design, especially if multi-core is involved. In addition, implementing a modern-sized cache with megabytes of SRAM storage will take a fairly large and expensive FPGA. Designing a memory controller for modern DDR4 DRAM is complex challenge; often an FPGA board will come with a built-in DRAM interface, but it is probably not possible for students to design one from scratch.

2.4 Cache Simulators

Software cache simulators are usually easier to design than FPGA ones. Complicated hardware structures often map into just a few lines of code. The primary downside of software cache simulators is that they are much slower than real hardware. A typical “cycle-accurate” CPU and memory system simulator, such as Gem5 [7] might have a slowdown of 1000x over native execution. Gem5 includes a simple DRAM simulator, but hooking up a more accurate DRAM simulator such as MemSys slows it down by nearly 2x [24]. Another DRAM simulator, DRAMSim2 [33] increase the runtime of an already slow simulator MARSSx86 by 30%.

Aside from speed, software simulation can have other issues. Software cache simulators are not always validated [41], and when they are, the accuracy can vary wildly [15]. The slowdown found when using software simulation means typically only trivial examples can be run, as otherwise even a small benchmark might takes days to weeks to run. Simulator codebases also tend to be complicated, so while changing cache parameters for experimentation is often straightforward, any more complicated experimentation with the entire memory hierarchy quickly leaves the range of class-project complexity and becomes a MS or even PhD thesis amount of work.

2.5 Operating System Exploration

One last way of investigating memory is the one we propose in this paper: investigating at the operating system level. The operating system is responsible for configuring the memory system, and can be deeply involved in its functioning, especially when virtual memory is involved.

As of version 4.17, the Linux *mm* memory-management directory has almost 130,000 lines of C code, not counting the additional per-architecture directories that average a few thousand lines each. This is complex code that is difficult to understand, even by the experts responsible for maintaining it.

The primary issue with understanding the code is it is designed to be highly performant at the expense of readability. High-end modern architectures, such as ARM and x86, can have complex virtual memory setups. The need for cross-platform code code can lead to multiple layers of abstraction that can be hard to follow.

To provide a teaching environment where students can learn about memory system and operating system interactions we propose using the simple vmwOS operating system, targeted at low-cost Raspberry Pi development boards. This allows experimentation on relatively simple systems, with a bare-bones and simple operating system. Using a low-cost development board helps; if something goes wrong it is inexpensive and simple to start over from scratch without risking damaging or corrupting a more important desktop or workstation. Also, by the time the students make it to an advanced memory class they typically have experience with the Raspberry Pi class boards and so have at least some knowledge of the hardware involved and a feel for what performance can be obtained from the boards.

3 RELATED WORK

There are numerous existing educational operating systems available. Ours is unusual in that we wish to make it easy to explore modern developments in the memory hierarchy.

3.1 Educational Operating Systems

Andrew S. Tanenbaum's Minix [35, 36] is a UNIX-like operating system used for teaching OS design. He also wrote Amoeba, designed for exploring distributed systems [37].

Another popular teaching operating system is xv6 from MIT [26] based on Version 6 of UNIX. There is a Raspberry Pi port of xv6 [18] but it does not currently have multi-core support. Another educational OS developed at MIT is JOS which primarily runs on x86 systems [25].

Xinu [12] is an educational operating system that has recently been ported to the Beaglebone Black embedded board.

NACHOS (Not Another Completely Heuristic Operating System) [10] is an educational OS but it only runs on MIPS processors. Pintos [28] was designed as a replacement to NACHOS that runs on x86 hardware. There is a Re-Pintos [21] effort that has plans to port it to the Raspberry Pi.

3.2 Bare-metal Raspberry Pi

There are many simple bare-metal operating systems written for the Raspberry Pi. Many of them are just people experimenting with the platform, and they rarely get to the stage of having a full traditional operating system with multi-tasking, filesystems, and device drivers. Most were written for testing or curiosity reasons and are not specifically designed for use in an educational environment.

One fairly complete embedded type operating system is Ultibo [2], written in Free Pascal. While it has fairly comprehensive device driver support, its memory support is primarily a matter of setting up the caches and virtual memory so simple programs can run.

BakingPi [1] is a bare-metal operating system development kit, designed for use in classes. It walks through a number of lessons on setting up a bare-metal Raspberry Pi operating system, but stops before reaching the program loading and multi-tasking stages.

For most of these projects virtual memory is something to be set up just enough so that caches work correctly, and then ignored. It is not an integral part of the operating system experience like we are attempting with vmwOS.

4 THE RASPBERRY PI ARCHITECTURE

The Raspberry Pi is a family of low-cost ARM boards developed by the Raspberry Pi Foundation [30]. These boards were originally designed for use in the educational market, but have also found great success in other areas. They are widely used by electronics hobbyist and as a base for general purpose low-cost computing. While low-performance by today's standards, the boards are perfectly capable of being used as desktop computers, and one common use is to make small-scale low-cost computing clusters [11].

There have been a large number of Raspberry Pi models since the release of the original in 2012, as seen in Table 1. We will focus in describing the hardware found in the Raspberry Pi3, as it has been out long enough to be commonly in use by students, and also because it is a 64-bit system and thus more interesting for the type of explorations we wish to undertake. Some selected details on some of the models are broken out in Table 2.

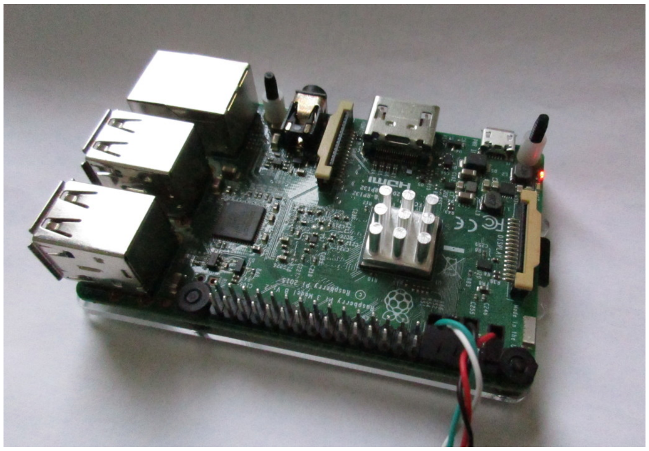


Figure 1: Raspberry Pi 3B running vmwOS, with serial console cables connected.

4.1 Raspberry Pi 3

The Raspberry Pi 3B is the first 64-bit iteration of the Raspberry Pi line. Recently the Pi 3B+ has been released, with the primary improvements being in thermal handling, moving the ACT (activity) LED back to a normal GPIO pin, and the upgrade to gigabit Ethernet (still limited by being driven by a USB2 connection). We will primarily be describing the Pi 3B (not the 3B+) in most of our discussions. A picture of a Pi 3B can be seen in Figure 1.

4.1.1 CPU. The CPU in a Pi 3B is a Broadcom BCM2837 (also known as a BCM2710). It is a quad-core 64-bit Cortex-A53 ARMv8 processor. It can run at 1.2GHz but often due to thermal reasons it is automatically scaled down to 600MHz when under load.

4.1.2 GPU. The GPU found on Raspberry Pis is the Broadcom VideoCore IV. The BCM2835 family was originally designed as a large GPU with an ARM CPU tacked on the side. Because of this, unlike most ARM setups, it is the GPU that is responsible for booting the system. The GPU even runs its own operating system (ThreadX). On the Pi 3B the GPU is capable of up to 28.8GFLOPS and has built-in video decoding hardware. The system RAM is shared between the GPU and CPU, with the RAM split set at boot.

The architecture of the GPU is not well known. Some work has been done to reverse-engineer the assembly language of the GPU [9, 16]. Some more information has reached the community after Broadcom hired Linux developer Eric Anholt to produce a fully functional Linux driver [22]. Linux supports the Quad-Processor Unit (QPU) interface that allows a simple GPGPU like interface to the GPU for calculations [17, 27].

4.1.3 Memory Hierarchy. The board contains 1GB of LPDDR2 RAM. One complaint about the Pi platform is the relative lack of

Table 1: Raspberry Pi Models: CPU type, cost and power usage when idle and under Linpack load. Power usage marked unknown is because we do not have a board of that type to test.

Model	CPU	Arch	RAM	Power Idle	Power Linpack	Cost
Model Zero	BCM2835	ARMv6	512MB	0.8W	1.4W	$5
Model Zero-W	BCM2835	ARMv6	512MB	0.6W	1.0W	$10
Compute Node	BCM2835	ARMv6	512MB	1.9W	2.2W	$30
Compute Node 3	BCM2837	ARMv8	1GB	?	?	$30
Model 1A	BCM2835	ARMv6	256MB	?	?	$25
Model 1A+	BCM2835	ARMv6	512MB	0.8W	1.0W	$20
Model 1B	BCM2835	ARMv6	512MB	2.7W	3.0W	$35
Model 1B+	BCM2835	ARMv6	512MB	1.6W	1.9W	$25
Model 2B (v1)	BCM2836	ARMv7	1GB	1.8W	3.6W	$35
Model 2B (v1.2)	BCM2837	ARMv8	1GB	1.7W	4.1W	$35
Model 3B	BCM2837	ARMv8	1GB	1.8W	4.8W	$35
Model 3B+	BMC2837	ARMv8	1GB	2.6W	7.3W	$35

Table 2: More detailed specifications for a selection of Raspberry Pi models

	Pi Zero	Pi 1B+	Pi 3B
CPU	BCM2835	BCM2835	BCM2837
Arch	ARM1176	ARM1176	ARMv8
Speed	1GHz	700MHz	1GHz
Cores	1	1	4
Bits	32	32	64
GPU	VideoCoreIV 250MHz	VideoCoreIV 250MHz	VideoCoreIV 300MHz
Memory	512MB	512MB	1GB
RAM type	LPDDR2	LPDDR2	LPDDR2
L1 icache	16k	16k	32k
L1 dcache	16k	16k	32k
L2 cache	128k (GPU)	128k (GPU)	512k
Network	none	100MB (USB)	100MB (USB)
Cost	$5	$25	$35

memory. According to Eben Upton (founder of the Raspberry Pi Foundation) this is not due to the architecture of the chip (although the design of the SoC addressing blocks makes it hard to add more RAM in a backwards compatible way). Rather the lack is due to the current relatively high cost of RAM [31].

The L1 instruction cache is 32k, 2-way, and 64-bytes wide, VIPT, pseudo-random replacement, critical word first. The L1 data cache is 32k, 4-way, and 64-bytes wide, PIPT, with pseudo-random replacement. Both caches are protected by error detection and correction. The L2 cache is 512k, 16-way, and acts primarily as a victim cache. There is a hardware prefetcher, as well as software prefetch instructions. Memory coherency uses the MOESI protocol.

The MMU supports multiple page sizes. There is a two-level TLB: a micro-TLB with 10 entries and a 512 entry 4-way main TLB. There is also a walk cache and an IPA cache (the latter is used when translating virtual-machine pages). The micro-TLB can be configured for round robin or random replacement.

4.1.4 Power Consumption. The Pi started out as a development board designed to be powered with a USB-micro connector. Because of this, early models did not draw much more than can be provided by at USB2.0 connection (500mA * 5V = 2.5W). For the 3B model, a 2.4A power supply is recommended and the power draw can exceed 5W. Table 1 lists power measurements for some models while idle and also while running the high-performance Linpack (HPL) benchmark.

4.1.5 Thermal Issues. As mentioned previously, the chip package was originally designed as a GPU board with a small CPU tacked to the side as a helper. Because of this the thermal sensor used for throttling was located over the GPU. As the CPU has been upgraded over the years this has led to issues as the CPU can overheat under load before the heat diffuses over to the temperature sensor. This leads to problems, such as the model 3B crashing when under heavy load, for example running high-performance Linpack [32]. The overheating issues have been addressed in the new 3B+ model.

4.1.6 Hardware Performance Counters. The Raspberry Pi has various hardware performance counters that can be used for architectural exploration. As with most modern CPUs, the counters involved vary with hardware generation. The Pi3B supports over thirty different events, including TLB, L1 instruction, L1 data, and L2 combined metrics.

4.1.7 Hardware Interfaces. While not necessarily useful for memory experiments, the SoC supports a large amount of other hardware interfaces. This includes i2c, SPI, camera, GPIO, PWM, HDMI, composite, USB, Ethernet, wireless, Bluetooth, serial ports, and an SD-card interface. See the Broadcom 2835 Peripherals Document for more information [8].

5 VMWOS BACKGROUND

vmwOS is an operating system written for the Advanced Operating System course at our institution. Its initial design goal was to allow hands-on coding in a generic senior or graduate level Operating System course.

In this course, the students gradually implement features. They start in week one with a blinking LED, and add features in following assignments until they end up with a limited multi-tasking operating system. There is an open-ended final project where more advanced topics not covered by previous assignments can be explored.

vmwOS is the total, working OS from which the class assignments are based. While it might be nice to just hand the students the ARM Architectural Reference Manual and the Broadcom Peripheral Document and have them figure things out on their own, that is not really practical for a class of this scope. Therefore solutions are provided based on the vmwOS code so that students who might struggle with one assignment do not fall completely behind as the course moves on.

Over the years the course has been taught the vmwOS has accumulated enough features that it is a fairly usable standalone OS. We have started looking into using it for other purposes, including the topic of this paper which is to adapt it for use when teaching a graduate level advanced memory system topics course.

One might suspect the name of the OS has something to do with virtual memory or virtual machines, but actually it is not-so-modestly named after the initials of the primary author's name. A screenshot of the system soon after boot running a multitasking demo can be seen in Figure 2.

The current goal of vmwOS is to provide a fully usable, multi-core, 64-bit operating system, that provides highly configurable access to the memory hierarchy. As an extra challenge the code should be kept simple enough that it can be used for teaching a senior-level operating systems course.

5.1 vmwOS: current status

vmwOS is under constant development, what follows is the current status of the operating system.

64-bit support

vmwOS was initially developed on the original 32-bit Raspberry Pi systems. It currently runs on all models of Pi, but in 32-bit mode. Even though the Pi 3B has an ARMv8 64-bit processor, vmwOS currently runs in ARMv7 32-bit compatibility mode. It should be relatively straightforward to update to native 64-bit. All of the driver work has already been done, but some difficult parts remain (including converting all of the assembly language to ARM64 which can be very different from ARM32 assembly).

Multi-core support

Multicore support is partially implemented and will hopefully be completed in the near future. At boot all four cores are configured, and inter-core communication via inter-process interrupts (IPI) is working. The remaining tasks are ensuring proper locking everywhere, and to make the scheduler multi-core aware.

Virtual Memory support

Currently vmwOS activates the MMU (memory management unit) and sets up some rudimentary kernel/user memory protection. However, full virtual memory at the page level is not enabled. Instead page tables are set up using coarse 1MB "section" pages, with a 1:1 mapping of physical to kernel addresses. There is memory protection between the kernel (which lives in the bottom 16MB) and the rest of userspace, but there is no inter-process protections.

This means the kernel currently behaves like MMU-less Linux (ucLinux) [39]. Binaries are position-independent and memory protection between processes is on the honor system.

Scheduler/Multi-tasking

The OS supports multi-tasking, including running jobs in the background (`waitpid()`). The scheduler is simple, using a round-robin algorithm. If no jobs are ready then an idle task runs, which enters low-power mode via the `wfi` instruction. A system timer is programmed to trigger the scheduler at 64 Hz.

Wait Queues

When a process blocks on I/O, it is put to sleep by being placed on a wait queue. When the I/O completes, all waiting threads are woken up.

Memory Allocation

The kernel memory allocation is very simple. There are two regions, user and kernel. Memory is handed out in 4k chunks, with a find-first method. A free list bitmap tracks the memory.

Executables

Executables use the Binary Flat (BFLT) file format [38]. Code is first built into regular ARM Linux ELF executables and an included custom tool is used to convert it to the much simpler BFLT. While BFLT is simple and much easier to implement than ELF, it does support more advanced features, such as limited support for run-time linking and shared libraries. Our OS currently does not support these more advanced features.

Userspace

Userspace executables for the OS are written in C, and a very simple C library (vlibc) is provided. This library provides the minimum support required for the included programs, which include a simple shell and various system utilities and games.

While generic Linux-like C programs can be ported fairly easily, there are some limitations that can make this difficult (a limited C library, variables/functions should be declared static to avoid use of the GOT, etc).

Device Drivers

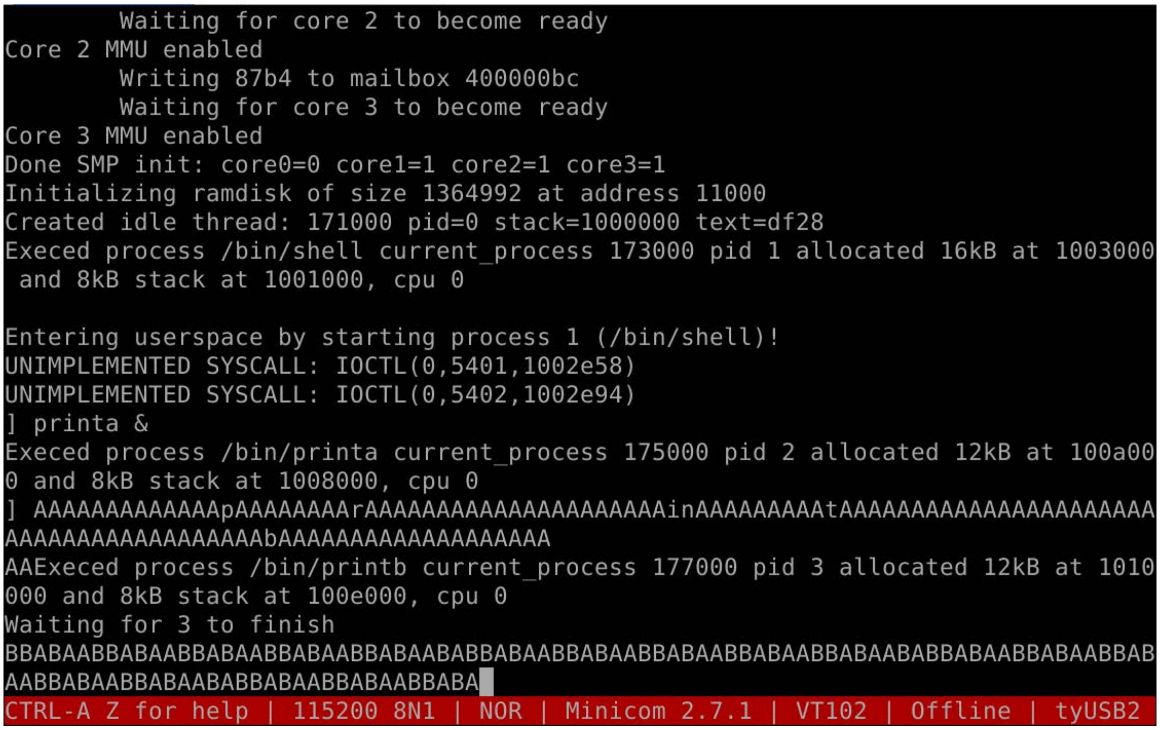
The Raspberry Pi supports a large number of devices on the SoC, but currently only a few have drivers in our OS. There are drivers for console output, both to the serial port as well as a framebuffer. Console input is supported over the serial port, as well as an external PS/2 keyboard adapter via GPIO (there is no USB support currently). There are drivers for the random number generator and temperature sensor. A simple sound beep effect is implemented via the pulse-width modulation (PWM) GPIO support. The only block device currently supported in a ramdisk, but work on an SD-card via SPI driver is underway.

Filesystems

Currently the only filesystem supported is romfs. The operating system currently boots with a read-only romfs ramdisk image. We plan to implement FAT and ext2 support in the near future.

5.2 vmwOS Challenges

Writing the vmwOS has been challenging, mostly due to lack of documentation. The Pi hardware is documented in the BCM Peripherals document [8] and the newer Pi 3 features can be found in the Quad-A7 document [40]. The ARMv7 processor is described in



```
        Waiting for core 2 to become ready
Core 2 MMU enabled
        Writing 87b4 to mailbox 400000bc
        Waiting for core 3 to become ready
Core 3 MMU enabled
Done SMP init: core0=0 core1=1 core2=1 core3=1
Initializing ramdisk of size 1364992 at address 11000
Created idle thread: 171000 pid=0 stack=1000000 text=df28
Execed process /bin/shell current_process 173000 pid 1 allocated 16kB at 1003000
 and 8kB stack at 1001000, cpu 0

Entering userspace by starting process 1 (/bin/shell)!
UNIMPLEMENTED SYSCALL: IOCTL(0,5401,1002e58)
UNIMPLEMENTED SYSCALL: IOCTL(0,5402,1002e94)
] printa &
Execed process /bin/printa current_process 175000 pid 2 allocated 12kB at 100a00
0 and 8kB stack at 1008000, cpu 0
] AAAAAAAAAAAAApAAAAAAAArAAAAAAAAAAAAAAAAAAAAAinAAAAAAAAAtAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAbAAAAAAAAAAAAAAAAAAA
AAExeced process /bin/printb current_process 177000 pid 3 allocated 12kB at 1010
000 and 8kB stack at 100e000, cpu 0
Waiting for 3 to finish
BBABAABBABAABBABAABBABAABBABAABABBABAABBABAABBABAABBABAABBABAABABBABAABBABAABBAB
AABBABAABBABAABABBABAABBABAABBABA
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | tyUSB2
```

Figure 2: vmwOS running a printa/printb multitasking demo in a serial console.

the Architectural Reference Manuals [3, 4], and more specific documentation can be found in Cortex-A53 related documents [5, 6]. Despite all of these rather large documents, often the best resource has been asking in the “bare-metal” section of the official Raspberry Pi Foundation forums [29].

Challenge: Booting

Booting is the first challenge. We had working boot code for earlier Pi models, but recent firmware versions changed to cause multicore to come up in hypervisor mode. This required changing the boot code so it starts off by switching back to normal mode. Previously system information was provided at boot via the old Linux ATAGS interface. Recently the Pis have transitioned to using device-tree files which involves having a full device-tree parser which is run early in the boot process.

Challenge: MMU

Setting up the MMU and page tables are complicated at the best of times. On the Pi you must have the MMU going before caches can be enabled (so you can mark the memory-mapped I/O regions as non-cacheable). There are about 20 different bits that need to be set properly in the multi-core setup before virtual memory happens properly. Even more of a challenge is getting the bits set to properly enable cache coherency between cores. Other issues can get in the way, such as what happens if some cores are accidentally left in hypervisor mode while the primary core is not. That particular problem led to a lot of frustration during bringup of multi-core support.

Challenge: Locking

On the Pi3 you need to use the load-link/store-conditional `ldrex` / `strex` instructions for locking. These instructions only work if the caches are running, and those need the MMU running first. This does make it difficult bringing up the system, as all of the above has to be working before locking can be used. Having proper locking in the serial I/O console can be difficult, especially if you use the same routines for printing early (pre-cache setup) boot messages.

Challenge: Cache Flushing

The ARM processor has various ways for flushing caches, and these become important once multi-core support is enabled. The documented ways for flushing the cache do not seem to always work. The best place to find working cache flushing code is the Linux kernel, as that is known to work and is often written by ARM engineers. Additionally, the Pi has a complex relationship with the GPU and various non-cacheable memory address ranges for communicating to the GPU via a mailbox interface. It is sometimes necessary to flush the caches after these interactions too, again this is not well documented.

6 PROPOSED MEMORY EXPLORATIONS

The goal of our vmwOS work is to provide an easy platform for operating system and memory hierarchy exploration. What follows are some research areas of interest that can be investigated with our setup.

6.1 Cache Experimentation

One relatively simple topic to explore is the behavior of a modern cache hierarchy as various parameters are manipulated.

One simple thing to explore is how a modern system behaves is how a system behaves with various parameters of the cache subsystem manipulated. Not all hardware supports low-level cache manipulation, and on most other operating systems it can be difficult if not impossible to change parameters once the system is up and running.

6.1.1 Disabling the Cache. The various Raspberry Pi models support disabling the caches. In fact, on some models the Pi the firmware does not enable caches at boot so it is up to the operating system to enable them. On older models the L1 data and instruction caches (and the related branch predictor) could be enabled separately.

Enabling the caches can make a huge performance difference. An example of the difference found with a `memset()` benchmark (on an older model 1A+ system) can be seen in Table 3. These are result from a homework assignment given to the students, where they time various `memset()` implementations with caches enabled and disabled. The students are often surprised by the huge increase in performance. They also get to implement their own `memset()` to see if they can beat the provided one.

This type of experimentation is more difficult when running in multicore mode, as the `ldrex` / `strex` locking used for mutexes depends on caches being enabled. It is still possible to do the experiment, but only if multicore is disabled.

6.1.2 Disabling the Prefetcher. The various Pi models support disabling/enabling the hardware memory prefetcher. Also some of the prefetch parameters can be tuned.

We have previously attempted disabling the prefetcher on other ARM development boards (such as the Cortex-A9 Pandaboard) under Linux and ran into many difficulties. In one instance the prefetcher was disabled and when we inquired were told the prefetcher hardware was broken on this particular device. In other cases when trying to disable / enable the prefetcher by setting the low-level CPU register from Linux it was blocked, as the TrustZone security setup by the firmware disabled changing the setting after boot.

We are interested in testing the prefetcher setup under the Pi in hopes it is more configurable than on the other boards we have tried. These experiments can also involve the ARM software prefetch instruction which is available on the Pi.

6.2 Virtual Memory Experiments

Virtual memory is a complicated topic that is not often well understood, even by computer architects. We find that one can learn a lot by trying to set up virtual memory from scratch, rather than relying on an existing operating system to do all the work for you.

6.2.1 Changing the Page Size. vmwOS currently only supports 1MB section-style paging, but once we add support for finer granularity paging it should be possible to do interesting explorations of the tradeoffs with multiple page sizes, or even transparent huge paging (THP). The Cortex A53 supports 4KB, 64KB, 1MB, 2MB, 16MB, 512MB, and 1GB page sizes.

In addition, once 64-bit support is finished, the large virtual address space (currently 48-bits) could be explored and used in non-traditional ways. One example could be exploring the overhead of address-space layout randomization which is used for security.

6.2.2 Page Protection. The ARM page tables support the normal sets of protections, including non-executable pages. The Cortex-A53 support kernel non-execute support, so that kernel execution cannot be redirected into userspace code. It would be interesting to test the feasibility of enabling this security feature.

6.2.3 Separate Coherence Zones. On the Cortex-A53 you can independently configure the cores to not take part in cache coherency. This could in theory allow different zones with separate views of main memory. It would be interesting to see how this feature could be made useful. In addition, cache-coherency can be specified at a per-page level via a bit in the page table entry.

6.2.4 TLB Performance. The TLB is multi-level and supports using address-space identifiers (ASID) to avoid the cost of flushing the TLB on context switch. Experiments can be done here to see what the tradeoffs are in using this.

Also the TLB on some of the Pi models is configurable, and the page replacement algorithm can be changed on the fly.

6.3 DRAM

The Pi3B comes with 1GB of LPDDR2 memory. We are somewhat limited in what we can do by the built-in memory controller and the soldered-on DRAM chip (in some cases it is bonded directly to the processor!)

6.3.1 DRAM Configuration. The firmware configures the DRAM parameter at startup. The firmware is just a file on the boot SD-card that is run by the GPU at boot time, and efforts have reverse-engineered some of how this works. In theory it might be possible to re-write the firmware to allow arbitrarily setting the settings of the DRAM in order to experiment with the timings.

6.3.2 Rowhammer. With low-level OS access to the memory, it would be interesting to see how hard it would be to trigger Rowhammer [20] style memory corruption. This should be made easier by the 1:1 virtual/physical mapping of our memory, as well as the ability to disable the caches.

6.4 NVRAM

Operating systems with volatile RAM typically operate in a perpetual state of potential failure. Any small fluctuation in power can wipe the memory, causing the data to be permanently lost. This can increase the overhead of any routines that wish to safely write to memory, as any dirty data must be written back to disk to be saved even if it remains resident in memory. There has been increasing interest recently to avoid this issue by using non-volatile RAM (NVRAM) in systems. While this is not necessarily a new development (old systems with core memory behaved just like this), modern systems typically assume system RAM is volatile and will lose its contents when power is removed.

Modern operating systems are not designed with NVRAM in mind and may not handle the addition of it to a system optimally. The Linux developers, for example, have had many discussions about how to support NVRAM without needless copying of memory or having to modify or create new filesystems [13].

There are various concerns on how to present NVRAM via the OS. One is to simply implement a filesystem on top of it, but current

Table 3: `memset()` **performance on Raspberry Pi A+ BCM2835 700MHz LPDDR2 RAM. Each test is 16 repetitions of a 1MB** `memset()`**. 1-byte is just a simple loop, one byte at a time, 4-byte is a simple loop, writing an integer at a time, 64-byte uses the ARM** `stm` **instruction to write 64B at a time. Results are shown with no cache, and various combinations of instruction cache, data cache, and branch predictor being enabled. Theoretical maximum speed of LPDDR2@400MHZ = 8GB/s.**

Implementation	Hardware Enabled	cycles	time	MB/s
C 1-byte	No Cache	937M	1.338s	12 MB/s
	L1-I$	355M	0.507s	32 MB/s
	L1-I$+brpred	271M	0.387s	41 MB/s
	L1-I$+brpred+D$	116M	0.166s	96 MB/s
C 4-byte	No Cache	206M	0.294s	54 MB/s
	L1-I$	68M	0.097s	165 MB/s
	L1-I$+brpred	64M	0.091s	176 MB/s
	L1-I$+brpred+D$	29M	0.041s	391 MB/s
ASM 64B	No Cache	23M	0.0335s	478 MB/s
	L1-I$	18M	0.0253s	631 MB/s
	L1-I$+brpred	18M	0.0257s	622 MB/s
	L1-I$+brpred+D$	9M	0.0126s	1268 MB/s
Linux / glibc				1400 MB/s
Theoretical max				8000 MB/s

filesystems assume high-latency accesses over various busses to disk (rather than a direct memory connection). The OS filesystem layer is not optimized for high-speed access and can squander the low-latency benefits of NVRAM. Another issue is that typically when accessing data from disk, copies are made to memory (in disk caches, or just to bring programs in to execute them). These extra copies provide no benefit on NVRAM systems, so the OS should be modified to avoid them either by turning off disk caches and by allowing execute-in-place of binaries. A final issue with NVRAM is consistency, to ensure that once a processor writes to NVRAM the value is stored out properly in case of a crash. Processor caches will cache memory accesses, and will require explicit cache flush instructions to make sure values get written out to NVRAM.

All of these NVRAM / OS interactions can be investigated with vmwOS, with the limitation that the Raspberry Pi systems themselves cannot directly support NVRAM. We propose emulating NVRAM in software to enable this type of experiment.

6.4.1 More NVRAM Background. NVRAM is a storage medium that retains stored data even when power is removed, unlike more traditional SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory). Until recently, no NVRAM options existed that could compete with SDRAM at the larger sizes, but now there are several technologies that will soon compete on the consumer market. These types of NVRAM include PRAM (Phase-change RAM), F-RAM (Ferroelectric RAM), and MRAM (Magnetoresistive RAM). Each of these have their respective drawbacks, but are likely to be competitively priced versus traditional SDRAM. They also will be able to compete in terms of size, write-erase cycles, and speed. The availability of NVRAM to consumers can provide interesting benefits to computing, and will trigger significant redesign in how software and operating systems manage memory.

6.4.2 Drawbacks of Modeling NVRAM using SPI Flash. The hardware being used is currently a Raspberry Pi 3 Model B+; chosen simply because it is the newest Raspberry Pi model. vmwOS supports the Pi 3B+ in all avenues that are needed for the research.

NVRAM is currently difficult to obtain, especially in sizes over a few Megabytes, and generally utilizes a specialized bus that is not available for external connection on a device such as the Raspberry Pi. In order to model NVRAM, an SPI (Serial Peripheral Interface) connected microSD card is being used as a backing store to provide permanent non-volatile storage. This is useful as an emulation tool as it is cheap, easily to interface, and easily available.

We connect an SD card to our system using an Adafruit SPI microSD breakout seen in Figure 3. SD cards have a startup command sequence that can be used to set the card to SPI mode, which must be done at 100-400kHz clock speed. Commands can then be sent to check the SD cards type and version to verify the card's available features and commands.

There are various drawbacks of using SD to emulate NVRAM.

Challenge: Lack of Byte Addressing

A common factor of the new NVRAM technologies is their ability to be byte addressable. This allows reading and writing with very little overhead in terms of manipulating data that is currently unnecessary. Unfortunately, in general SD cards are only able to be read and written in blocks, commonly of sizes 512 or 1024 bytes. This means that if a single byte needs to be updated, the entire 512 byte block must be read out, changed, and then written back. Typical program behavior of repeatedly writing to small local variables in RAM can lead to huge overheads and waste if these are directly mapped to SD flash.

Challenge: Speed

One drawback of the microSD storage is significantly slower speeds than SDRAM, and slower than the NVRAM begin simulated. For example DDR4 PC4-25600 SDRAM, which is a fairly common memory used in consumer available computers, has a theoretical maximum bandwidth of 25.6GB/s. The current testing has the microSD communicating SPI with a clock rate of 117.37kHz. Neglecting overhead in sending commands, this gives a theoretical

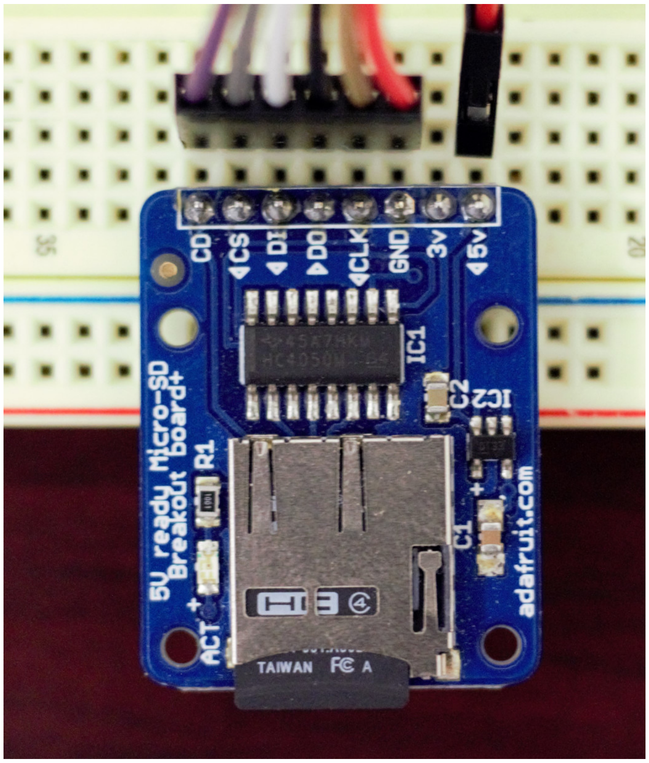


Figure 3: Adafruit microSD breakout module used for NVRAM modeling

maximum bandwidth of 14.671KB/s meaning the SDRAM is theoretically 1.75x10^6 times faster. Increasing the SPI clock speed to 25MHz, which is generally an upper limit for high performance SPI communications with an SD card, the SDRAM would still be 8192 times faster.

This holds for the latency for access as well. Most SD cards must be read and written in 512 byte chunks. The latency to read/write a byte is minimum the command size plus the time to read a byte, and maximum command size and time to read 512 bytes. At 117.37kHz this gives 477us-35.3ms. Compare this to the average latency of SDRAM, which is generally in the range of a few nanoseconds.

Challenge: Direct Memory Mapping

In general computing, the CPU can directly read and write data in memory. This has low overhead and is extremely fast. When using the microSD card as NVRAM, the data cannot be directly accessed by the CPU, and instead needs to be read into the SDRAM. This adds overhead for the processes utilizing the NVRAM, and increases the complexity of the code.

Another issue is what to do with writes. One way of doing this is to mark all RAM as read-only and using the virtual memory subsystem to trap on writes, and then at this point write the memory to SD as well as memory (or alternately, to cache the write and write out when convenient or once enough dirty memory has accumulated).

Challenge: Limited Read Write Cycles

A computing system is more or less constantly accessing memory, whether to fetch instructions or to load and store data. Storing data is troublesome, as a common characteristic of flash based storage is a limit in write cycles. The more write cycles a certain sector has, the higher the likelihood of failure. For a Kingston MLC Flash, this is rated at 3000 write cycles. Using the max bandwidth at 117.37kHz clock speed for SPI this could be accomplished in 105.9s for a 512 byte section. A 2GB SD card would have around 3906 usable sections of 512 bytes, so even with wear leveling to distribute writes evenly across all available space, it would be worn out completely in 114.9 hours of continuous writes.

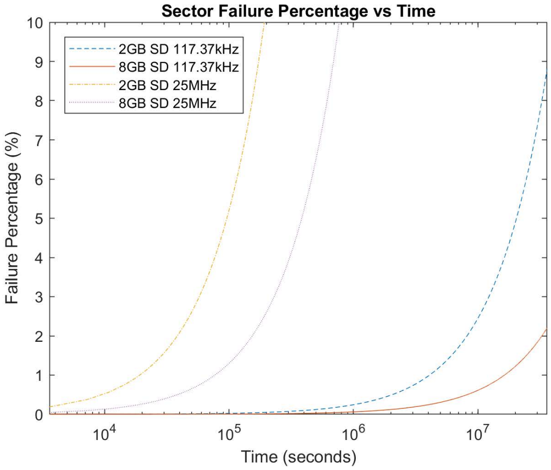


Figure 4: Sector failure percentage versus time under sustained write at 117.37kHz and 25MHz for a 2GB and 8GB SD card with 3000 writes per sector before likely failure

The largest issue with this isn't necessarily complete failure, it is that as sectors fail the chances of irrecoverable data loss greatly increase. This can create system instability as well as loss of important data. The percentage of sector failure over time based on SD size and write speed is shown in Figure 4.

6.4.3 Proposed SD/NVRAM Memory Emulation. There are various ways that NVRAM can be implemented in an operating system. We look at having a hybrid NVRAM/DRAM interface, where the system has both DRAM and NVRAM. User processes can request that regions of NVRAM be mapped into their address space, which are then accessed as normal RAM. We then emulate this using the virtual memory system and make sure that all write accesses are backed up to the SD. Upon reboot the OS restores these memory areas and processes can find them again.

Corruption of the memory allocation table can corrupt all of the stored NVRAM. For this reason, a fail safe was implemented adding an additional 512 byte buffer and 512 byte information block to the memory allocation table. Any time data is written to the memory allocation table it is first written to the buffer, a bit is set in the information block indicating the memory allocation table is about to be written, and the address to be written to is added to the information block. The bit in the information block is only cleared when the write to the memory allocation table has successfully finished. If the write fails after some bytes have already been written, but not all, or if the kernel boots and reads in the informational block and the write bit is set, then the kernel can recover without the corruption of data.

While this is not the only means of failure, it is one significant avenue that is closed off. Further protection would include backing up the memory allocation table, but depending on the amount of processes that have allocated data, this table could grow very large. These issues have already been extensively researched in the development of file systems, so the implementation of a true file system would provide much better reliability. As a compromise, there have been NVRAM specific file systems developed, such as NOVA [44], which attempt to work optimally in NVRAM systems.

6.4.4 Implementation for Userspace Programs. The SD card model of NVRAM relies on SPI communications, and SPI is implemented as a driver in the vmwOS operating system. This creates difficulties in utilizing NVRAM for kernel memory, especially during boot, but can be easily utilized for userspace programs. Anytime a userspace program needs to read information from the NVRAM, 512 bytes are read into a local buffer in SDRAM. This creates overhead vs SDRAM, but if the use of the local buffer is ignored it is easy to see how it is comparable to how NVRAM can eventually be utilized.

6.5 Other Potential Research Areas

6.5.1 Power and Energy. Power and energy are increasingly important in modern CPU and memory designs. Unfortunately the Pi does not provide an easy way to monitor the energy use, as this would be a great addition to any of the studies proposed here. We have conducted many power/energy studies of the Pi, but these require manually instrumenting the incoming power connection with a sense resistor and having an external device record the power consumption.

The Pi does support dynamic voltage and frequency scaling. It could be interesting to use this and see how it affects memory hierarchy performance. Unfortunately the DVFS interface is not well documented on the Pi hardware.

6.5.2 Hardware Performance Counters. The various Pi models have support for hardware performance counters, including various cache and TLB related events. It should be possible to read these out during system operation, and behavior of the system can be optimized on the fly based on live feedback of how the memory system is behaving.

6.5.3 GPU. The Raspberry Pi has a powerful GPU. It shares main memory with the CPU, and communication between the two is done via a mailbox interface. The GPU is not documented well, although Linux supports GPGPU-style programming via a “QPU” interface as well as full accelerated 3D graphics support.

It would be interesting to see if the GPU/CPU combination could be treated more like a heterogeneous system and design the operating system around this concept. One struggle is that unlike some newer high-end GPU designs, the GPU can’t take part in cache coherency. The other primary barrier is the lack of low-level GPU documentation, as currently most of it is reverse engineered.

6.5.4 Virtualization/Hypervisor. The Cortex-A53 has full virtualization support, and this can interact in complex ways with the memory hierarchy. The system provides a low-cost base for experimenting with virtual machines.

6.5.5 System Call Overhead. The CPUs currently found in all of the Pi Models (ARM1176, Cortex-A7, Cortex-A53) are in-order and thus not vulnerable to the recent Meltdown and Spectre vulnerabilities [23]. We can still investigate the overhead of the various mitigation methods, especially those that involve changing the cache behavior or moving the kernel into a separate address space.

Also, for meltdown-type fixes much of the overhead comes at system call time. It would be interesting to experiment with other methods of user/kernel communication that might have less overhead than the traditional system call interface.

6.5.6 Older Pi Features. Various older Raspberry Pi models and their ARM1176 CPUs have interesting features not found on the newer systems. This can provide compelling areas of research, if like us, you have large piles of the older models gathering dust.

One interesting feature is the availability of 16kB of Tightly-coupled Memory (TCM) which acts as a fast scratchpad. This does not seem to be supported by other operating systems available on the Pi. Another feature found on the older CPUs is the ability to disable the branch predictor, to configure the hardware prefetcher behavior, and to change the TLB replacement policy.

6.6 Missing Opportunities

Modern high-end CPUs, especially x86 ones, have features that would be fun to experiment with but support is just not in the ARM cores on the Pi.

This includes things such as encrypted memory and hardware bounds checking. It might still be possible to conduct experiments in these areas by faking support via an emulation layer, as we do with NVRAM support.

Another area is the addition of other memory types. We are limited to the DRAM provided with the board. It would be nice if we could somehow provide a custom memory controller, possibly via FPGA, and experiment with new and novel DRAM configurations. Currently though this is not within the capabilities of the board without a major intrusive redesign.

7 CONCLUSION AND FUTURE WORK

We describe vmwOS, a simple teaching operating system that runs on the low-cost Raspberry Pi series of embedded boards. We plan to use this OS in an advanced operating system class to introduce the many facets of the memory hierarchy found in modern computing hardware. Most operating systems have extremely complicated interactions with the memory management systems in modern CPUs, we wish to keep our operating system simple so the underlying interfaces can be more easily explored.

We plan to continue extending vmwOS so that it can become as complete of an operating system as possible while still striving to retain its simplicity. The most pressing needs are: full multicore support, 64-bit support, full read/write filesystem support, and block-device support. Once we have all of this working the vmwOS will be an ideal platform for teaching students about modern memory hierarchies.

All of the code for vmwOS can be found in our git repository: https://github.com/deater/vmwos.git

ACKNOWLEDGMENTS

Pascal's contribution was partially supported by the National Science Foundation under Grant No. SSI-1450122.

REFERENCES

[1] Baking Pi. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/.

[2] ultibo. https://ultibo.org/.

[3] ARM. *ARM Architecture Reference Manual ARM v7-A and ARM v7-R edition*, 2011.

[4] ARM. *Cortex-A7 MPCore Technical Reference Manual*, 2012.

[5] ARM. *Cortex-A53 MPCore Processor Technical Reference Manual*, 2014.

[6] ARM. *Bare-metal Boot Code for ARMv8-A Processors*, 2017.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood. The gem5 simulator. *Computer Architecture News*, 39(2):1–7, May 2011.

[8] Broadcom. *BCM2835 ARM Peripherals*, 2012.

[9] K. Brooks. rpi-open-firmware. https://github.com/christinaa/rpi-open-firmware.

[10] W. A. Christopher, S. J. Procter, and T. E. Anderson. The Nachos instructional operating system. In *In Proc. on USENIX Winter 1993 Conference*, pages 4–4, 1993.

[11] M. Cloutier, C. Paradis, and V. Weaver. A raspberry pi cluster instrumented for fine-grained power measurement. *Electronics*, 5(4):61, 2016.

[12] D. Comer. *Operating System Design — The Xinu Approach.* CRC Press, second edition, 2015.

[13] J. Corbet. Supporting filesystems in persistent memory. *Linux Weekly News*, Sept. 2014.

[14] T. Gleixner and I. Molnar. Performance counters for Linux, 2009.

[15] A. Gutierrez, J. Pusdesris, R. Deslinksi, T. Mudge, C. Sudanthi, C. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2014.

[16] H. H. Hermitage. Tools and information for the Broadcom VideoCore IV (RaspberryPi). https://github.com/hermanhermitage/videocoreiv.

[17] H. H. Hermitage. videocoreiv-qpu. https://github.com/hermanhermitage/videocoreiv-qpu.

[18] Z. Huang. This is an xv6 port on Raspberry Pi 2 and 3. https://github.com/zhiyihuang/xv6_rpi2_port.

[19] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk.* Elsevier, first edition, 2008.

[20] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proc. 41st IEEE/ACM International Symposium on Computer Architecture*, pages 361–372, June 2014.

[21] L. Kurusa. Re-Pintos: Revitalizing an instructional OS. SOSP Poster Session, 2017.

[22] M. Larabel. Eric Anholt leaves Intel's Linux graphics team for Broadcom. *Phoronix*, June 2014.

[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[24] C. Menard, J. Castrillon, M. Jung, and N. Wehn. System simulation with gem5 and systemc: The keystone for full interoperability. In *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 62–69, July 2017.

[25] MIT. The JOS instructional operating system. https://pdos.csail.mit.edu/6.828/2016/overview.html.

[26] MIT. Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2012/xv6.html.

[27] mn416. QPULib: A language and compiler for the Raspberry Pi GPU. https://github.com/mn416/QPULib/.

[28] B. Pfaff, A. Romano, and G. Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 453–457, 2009.

[29] Raspberry Pi Foundation. Bare metal, assembly language. https://www.raspberrypi.org/forums/viewforum.php?f=72.

[30] Raspberry Pi Foundation. Raspberry pi. http://www.raspberrypi.org/.

[31] Raspberry Pi Foundation. Raspberry pi 3 model b+ on sale now at $35. https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/.

[32] Raspberry Pi Foundation Forums. Pi3 incorrect results under load (possibly heat related). https://www.raspberrypi.org/forums/viewtopic.php?f=63&t=139712&sid=bfbb48acfb1c4e5607821a44a65e86c5.

[33] D. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10:16–19, 2011.

[34] J. Schneider, J. Peddersen, and S. Parameswaran. A scorchingly fast FPGA-based precise L1 LRU cache simulator. In *19th Asia and South Pacific Design Automation Conference*, pages 412–417, Jan. 2014.

[35] C. Severance. Andrew S. Tanenbaum: The impact of MINIX. *Computer*, 47(7):7–8, July 2014.

[36] A. S. Tanenbaum. Lessons learned from 30 years of Minix. *Communications of the ACM*, 59(3):70–78, Mar. 2016.

[37] A. S. Tanenbaum and G. J. Sharp. The Amoeba distributed operating system.

[38] uClinux. BFLT binary flat format. https://retired.beyondlogic.org/uClinux/bflt.htm.

[39] uClinux. Embedded Linux microcontroller project. http://www.uclinux.org/.

[40] G. van Loo. ARM quad A7 control, Aug. 2014.

[41] V. Weaver. *Using Dynamic Binary Instrumentation to Create Faster, Validated, Multi-core Simulations.* PhD thesis, Cornell University, May 2010.

[42] V. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2013.

[43] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, Mar. 1995.

[44] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. 14th USENIX Conference on File and Storage Technologies*, Feb. 2016.