Investigating Performance Bottlenecks in the Plan 9 SPI Driver

Nicholas Ingravallo

March 9, 2026

Abstract

This paper aims to identify issues within the Serial Peripheral Interface (SPI) driver for the Plan 9 operating system developed by Bell Labs. In a multi-core environment under the Raspberry Pi 4 hardware, interfacing with the SPI driver on non-0 cores in a real-time environment resulted in a significant slowdown in peripheral performance. We demonstrate that these performance bottlenecks can be mitigated by implementing a direct memory polling solution for the SPI driver.

1 INTRODUCTION

The Plan 9 Operating System was developed by Bell Labs, released in 1992 for university use and again in 1995 for non-commercial purposes. Plan 9 was designed to be an improved Unix, primarily emphasizing a file-based system in a much more literal sense than the Unix implementation. In a study focused on scheduling in operating systems, this work takes a deep dive into the Plan 9 operating system, discovering how its scheduling operates alongside making improvements to one of its newer drivers on the Raspberry Pi 4 architecture.

The circumstances of this study arose from observing flickering in an LED matrix connected via the SPI controller to a Raspberry Pi 4 running Plan 9. Due to performance demands of the LED controller's code, the real-time scheduler was initially suspected of failing. Real-time tasks should maintain high priority and performance should not be diminished despite high operating system load. Further investigation led to discovering performance lapses in the Plan 9 SPI controller on the Raspberry Pi 4.

2 TECHNICAL BACKGROUND

Several different testing methodologies were developed to discover issues in the real-time scheduler. At first, a simulated system load program was run on each core while simultaneously running the real-time scheduler on each of the Raspberry Pi 4's cores. Then, a solution probing the GPIO driver with simulated load similar to the code in the LED matrix was utilized in the real-time scheduler loop under the control system load. Finally, an investigation into the SPI driver revealed a slowdown in a multi-core environment with non-0 cores suffering a significant reduction in performance. The cause of the performance degradation is then investigated, with a proposed alternative SPI-write polling-based solution.

3 SIMULATING REAL-TIME SCHEDULER SYSTEM LOAD

To stress-test the real-time scheduler, a C program with a calculated 30-second loop was implemented with a 1ms real-time delay. If the RTS functioned correctly, completing 30,000 loop cycles at a real-time delay of 1ms would lead to a roughly 30-second program runtime minus setup overhead. No slowdown should be observed despite simulated system load (a `while(1)` loop) on adjacent cores due to it prioritizing the real-time labeled task.

```
...
 pid = getpid();
    path = smprint("/proc/%d/ctl", pid);
    cfd = open(path, ORDWR);
    free(path);
    fprint(cfd, "wired %d\n", core);
    fprint(cfd, "period 1ms\n");
    fprint(cfd, "cost 1ms\n");
    admitVal = fprint(cfd, "admit\n");

    for (i = 0; i < 30000; i++) {
        sleep(0);
    }
    print("done\n");
...
```

Despite testing this code across all cores, there was no evidence of any slowdowns due to failure of the real-time scheduler. The LED matrix uses the GPIO and the SPI driver to set an LED pin state, so these drivers were called in our test loop to verify whether there was a degradation in performance.

```
...
 fdg = open("#G/gpio", OWRITE);
 if (fdg < 0) {
  perror("GPIO OPEN");
  exits(nil);
 }

 pid = getpid();
 path = smprint("/proc/%d/ctl", pid);
 cfd = open(path, ORDWR);
 free(path);
 fprint(cfd, "wired %d\n", core);
 fprint(cfd, "period 1ms\n");
 fprint(cfd, "cost 1ms\n");
 admitVal = fprint(cfd, "admit\n");
 print("admit val - %d\n", admitVal);

 spictl = open("#pi/spictl", ORDWR);
 spidat = open("#pi/spi0", ORDWR);
 if (spictl < 0 || spidat < 0) {
```

```
 perror("spi open");
 exits(nil);
}

memset(buf, 0x41, 8);
for (i = 0; i < 30000; i++) {
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 fprint(fdg, "set 25 1\n");
 pwrite(spidat, buf, 8, 0);
 sleep(0);
}
print("done\n");
...
```

The same simulated load tests were performed, and it was observed that cores 1-3 had a significant system performance reduction. With the simulated load removed, similar results (within 0.01s) were observed. To verify whether the GPIO driver or the SPI driver was the cause of the performance degradation, two different versions of the test code were written. The first test had the GPIO-write code enabled, while the second test had the SPI-write code enabled. The performance of the GPIO-only test showed no performance degradation, while the SPI-only code showed identical results to the dual GPIO-write and SPI-write code demonstrated previously.

4 THE SPI DRIVER

Below is a demonstration of the code that was used to probe the SPI driver in an attempt to replicate the slowdown, with results below the code block. Some code provided by [1].

```
...
pid = getpid();
path = smprint("/proc/%d/ctl", pid);
cfd = open(path, ORDWR);
free(path);
fprint(cfd, "wired %d\n", core);
fprint(cfd, "period 1ms\n");
fprint(cfd, "cost 1ms\n");
admitVal = fprint(cfd, "admit\n");
print("admit val - %d\n", admitVal);

spictl = open("#pi/spictl", ORDWR);
spidat = open("#pi/spi0", ORDWR);
if (spictl < 0 || spidat < 0) {
 perror("spi open");
```

```
  exits(nil);
 }
 memset(buf, 0x41, 8);
 for (i = 0; i < 30000; i++) {
  pwrite(spidat, buf, 8, 0);
  sleep(0);
 }
 print("done\n");
...
```

Table 1: Results Invoking SPI Driver

Attempt	Core 0	Core 1
Run 1	30.01s	69.68s
Run 2	30.02s	69.68s
Run 3	30.01s	69.67s

Results indicated a 133% slowdown upon invoking the SPI driver on cores other than core 0. Similar results were reported on cores 2 and 3.

The performance degradation suggested issues with cross-core interrupts. The existing driver utilized the Direct Memory Access (DMA) controller. While DMA allows for parallel processing, the overhead of setup and interrupt handling across multiple cores appeared to be the bottleneck. The target of our testing was within the `spirw` function.

A polling-based solution that bypasses the DMA controller was tested, where the SPI driver manually copies data to the SPI controller while utilizing loop guards. In this implementation, the new `spirw` code loops through the buffer data, copy its contents to the SPI controller, and utilize loop guards until transfers are complete.

```
void spirw(uint cs, void *buf, int len){
    int i;
    // ... initialization code unchanged ...
    spi.regs->cs = Rxclear | Txclear | Adcs | Ta;
    for (i = 0; i < len; i++) {
        spi.regs->data = ((char*)buf)[i];
        while ((spi.regs->cs & Txd) == 0);
    }
    while ((spi.regs->cs & Done) == 0);
    spi.regs->cs = 0;
    qunlock(&spi.lock);
    poperror();
}
```

5 COMPARATIVE RESULTS

The polling solution eliminated the multi-core slowdown and allowed for higher data throughput per real-time cycle. No interface slowdown or hanging due to the kernel directly processing the

SPI interface request was experienced, which was initially raised as a concern as the parallelism provided by the DMA controller was removed.

Table 2: Polling vs. DMA Performance

	Polling		DMA	
Run	Core 0	Core 1	Core 0	Core 1
Run 1	30.01s	30.02s	30.01s	69.68s
Run 2	30.02s	30.02s	30.02s	69.68s
Run 3	30.03s	30.02s	30.01s	69.67s

6 PERFORMANCE DEGRADATION CAUSE

Investigation into the Pi 4 system interrupt implementations in Plan 9 `src/9/bcm/trap4.c` revealed the cause for the slowdown experienced when interfacing with the SPI driver.

```
Nsgi = 16,  /* software-generated (inter-processor) intrs */
Nppi = 32,  /* sgis + other private peripheral intrs */
...
...
if (m->machno == 0) {
 cpumask = 1<<0;    /* just cpu 0 */
 for (i = Nppi; i < sizeof idp->targ; i++)
  idp->targ[i] = cpumask;
 ...
}
else {
 ...
 for (i = 0; i < Nppi; i++)
  idp->targ[i] = 1<machno;
 ...
}
```

Core 0 handles all peripheral interrupts, while software-generated interrupts are handled specifically on cores 1-3. This observation is relevant due to the DMA controller's interrupts, which are considered peripherals, occurring on core 0. If a program is running on core 1 with any code that directly or indirectly interfaces with the DMA controller, the DMA initialization will occur on our dedicated core. However, upon the call to `dmastart` in the `spirw` SPI write code, the interrupt enable would be handled on core 0.

```
void
dmastart(int chan, int dev, int dir, void *src, void *dst, int len)
{
 Ctlr *ctlr;
 Cb *cb;
 int ti;
```

```
 ctlr = &dma[chan];
 if(ctlr->regs == nil){
  ctlr->regs = (u32int*)(DMAREGS + chan*Regsize);
  ctlr->cb = xspanalloc(sizeof(Cb), Cbalign, 0);
  assert(ctlr->cb != nil);
  dmaregs[Enable] |= 1<regs[Cs] = Reset;
  while(ctlr->regs[Cs] & Reset)
   ;
  intrenable(IRQDMA(chan), dmainterrupt, ctlr, 0, "dma");
 }
}

static void
dmainterrupt(Ureg*, void *a)
{
 Ctlr *ctlr;

 ctlr = a;
 ctlr->regs[Cs] = Int;
 ctlr->dmadone = 1;
 wakeup(&ctlr->r);
}
```

The cross-core wakeup inside of dmainterrupt() causes the delay in the real-time scheduler.

`#define IRQDMA 16 #define IRQDMA(chan) (IRQdma0+(chan))`

IRQdma0 is defined as interrupt request number 16. The logic inside of intrenable() (defined as irqenable()) shows that all DMA interactions will be handled on core 0.

```
...We call intrenable() in dmastart: (dma.c)
intrenable(IRQDMA(chan), dmainterrupt, ctlr, 0, "dma");

....Which is defined as: (fns.h)
#define intrenable(i, f, a, b, n) irqenable((i), (f), (a))

...IRQLOCAL and IRQGLOBAL are defined here: (trap4.c)
//src: trap4.c
#define IRQLOCAL(irq) ((irq) - IRQlocal + 13 + 16)
#define IRQGLOBAL(irq) ((irq) + 64 + 32)
...
/*
 *  enable an irq interrupt
 *  note that the same private interrupt may be enabled on multiple cpus
 */
void
irqenable(int irq, void (*f)(Ureg*, void*), void* a)
{
 Vctl *v;
 int ena;
```

```
 static char name[] = "anon";

 /* permute irq numbers for pi4 */
 if(irq >= IRQlocal) // note: io.h:23: IRQlocal = 96,
  irq = IRQLOCAL(irq);
 else
  irq = IRQGLOBAL(irq);
 if(irq >= nelem(vctl))
  panic("irqenable irq %d", irq);


 ...
}
...
```

The interrupt request for DMA is considered a global interrupt request, which is calculated to be above request number 32, placing the interrupt handler on core 0. Therefore, when the SPI-write code is called using the DMA solution on cores 1-3, the interrupt requires a cross-core wakeup.

7 ALTERNATIVE SOLUTIONS

An alternative solution is proposed which would reroute the DMA controller interrupt. Using `intrto(int, int)`, a program calling the SPI driver, thus the DMA driver, would now have its respective driver calls occur on the original process' core. For example, a process running on core 2 would reroute the DMA interrupt to core 2, and a new process on core 3 would reroute the DMA interrupt to core 3.

The performance improvements were identical to the polling solution, with an expected 30-second runtime occurring for our test program across all cores. Consecutive and parallel runs across different cores also performed as expected. No side effects from this interrupt-based solution were reported in the usability testing of the operating system.

This proposed interrupt-based solution is a much more invasive change than our polling solution, but given its stability and speed it is worth consideration.

```
// dma.c
166  ...
167  intrenable(IRQDMA(chan), dmainterrupt, ctlr, 0, "dma");
168 }
->
-> intrto(m->machno, (IRQDMA(chan) + 64 + 32));
->
169 cb = ctlr->cb;
170 ti = 0;
171 switch(dir){
 ...
```

ACKNOWLEDGEMENTS

A special thank you to Brian Stuart for his guidance in this research. His expertise on the inner workings of Plan 9 and his overall support were a tremendous help.

References

[1] Brian Stuart. Source code for `ledmat.c`. 2024. Led Matrix Controller Code, Drexel University.

[2] Bell Labs. Plan 9 from Bell Labs Source. 2021. Available at `https://github.com/0intro/9legacy/tree/main`.