Analysis of SMP Kernel Scheduling

Jnaneshwar Weibel

April 21, 2018

Abstract

As algorithms become larger and more resource intensive, computer hardware must become increasingly efficient to effectively run such programs. In particular the capability for multiprocessing allows systems to distribute work across multiple CPUs. Modern operating systems and their respective schedulers have the opportunity to make policy decisions, such as load balancing and locality of access to shared memory, when distributing programs that can influence the performance of both the kernel and user programs.

1 Introduction

Hardware has evolved to become increasingly more complex and efficient, through areas such as processor frequency and memory latency. Amdhal's law gives diminishing returns on the impact of parallelizing computation. However, with respect to symmetric multiprocessing (SMP) in the kernel, Gustafon's law proves as now as a **scheduler?** can run "in the same time with a larger workload" [6]. Thus as new system architectures take advantage of these processors. They also require new approaches for system software and applications that run on top of this hardware [1].

Within an operating system, the task scheduler is responsible for running user tasks and determining an execution order. Yet with multiprocessing the scheduler offers an opportunity to distribute work across multiple cores. In a preemptive system, the scheduler may migrate tasks between different cores which generally will incur a local penalty in cache performance yet may improve global performance. Thus for this report several variations on the classical task scheduler will be compared to their impact on system performance and task scheduling.

A run-queue will consist of a FIFO list of tasks and will be protected by a run-queue specific lock. The performance of several tasks will be measured using a single global run-queue, a per-core run-queue without pull migration, and a per-core run-queue with pull migration. Pull migration refers to periodically re-balancing the load on two run-queues. In addition, some tasks will be constrained to a smaller set of CPUs and thus may be fixed to a single process or not undergo push migration. This paper will demonstrate the performance implications that various scheduling algorithms have on both local and global task execution.

2 Environment?

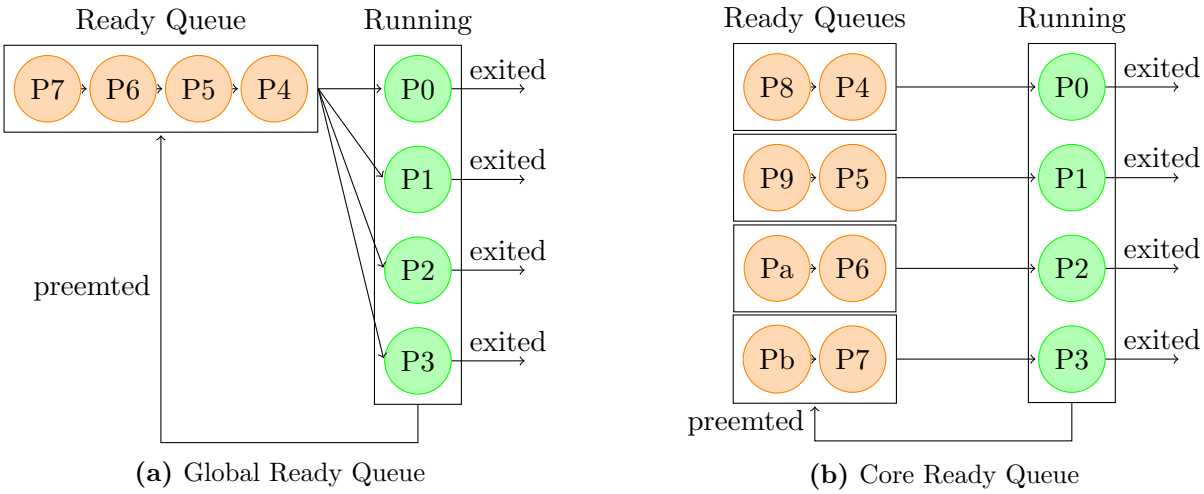
The OS will be compiled for the ARMv8 (aarch64) instruction set and will be executed on the Raspberry Pi 3 B. The device includes a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor where each core has a 16KB L1 data cache, a 16KB L1 instruction cache, and a shared 128KB L2 cache [2] [3]. The caches operate using the MOESI protocol with intra-core cache coherency and each have a fixed line length of 64 bytes in addition to a data side prefetch engine [4] [5]. The memory address space is configured using a linear two level translation table with accessible memory configured as normal outer and inner write-back, write-allocate and device register memory configured as device nGnRnE. Performance will be measured using the ARM Performance Monitor Unit (PMU) and the core timer. The timer executes at a frequency of 19.2 MHz [9] and each core maintains its own core timer which will preempt the current task each MS.

Both the kernel and pseudo-user processes will execute in EL1; however, the kernel will use SPx and the user processes will use SP0.

3 Definitions?

A ready queue can be defined as a generic data structure with two primary operations: ready and next. For its internal manipulation a ready queue is protected by a spinlock. Internally there are several queues corresponding to different priority levels allowing for processes with different priorities to be pushed onto the queues with no contention. However, for discussion we will assume that all processes run at the same priority.

A naieve implementation for a SMP scheduler involves a global ready queue which all cores pull processes from. This naieve implementation acts identical that in a single core system. On SMP systems this data structure is benificial since it guarantees equal distribution of workload across all cores in addition to implementation simplicity. Additionally, this workload distribution is not affected by a rapid stream of exiting processes. However, the approach suffers from kernel lock contention since all cores must synchronize access to the queue.



(a) Global Ready Queue **(b)** Core Ready Queue

Figure 1: Scheduler Structures

Under the current scheduler, a process, if eligible, is migrated during preemption. In general the scheduler will opt to keep a process on the same CPU unless another core is sufficiently empty or its affinity set dictates otherwise. In essence this rebalancing means that as processes are rescheduled, the cores will tend towards a balanced configuration. However, it is also possible that a core will have a stream of processes terminate leading to a load imbalance between ready queues (fig. 3).

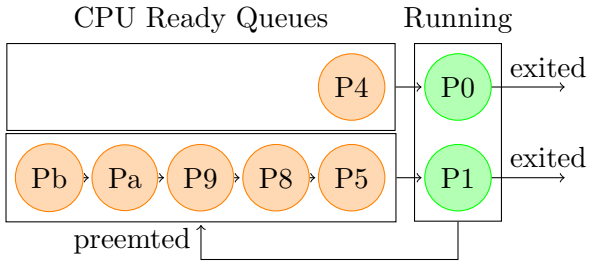


Figure 2: Unbalanced Scheduling

Pull migration is an effort to improve the re-balancing latency for queues which exhibit the behaviour in fig. 3. Periodically, an idle core will search for an overloaded core and pull eligible processes from the

tail of its queue. This task requires locking the busy queue for a longer period of time; while, the idle queue tends to have have its lock acquired more frequently. This task leads to high lock contention for both queues which can negatively impact the time spent in the kernel. Specifically, while iterating, the busy core will be unable to retrieve the next runnable process. Thus it is important to pick a large enough interval to do the re-balancing that it does not cause much strain on performance. However, in most cases, if a core is not sufficiently loaded the pull migration task will instead exit prematurely instead of locking both cores.

TODO: Process Fairness

4 Performance

Different user programs were tested against the variations of the OS scheduler described previously. In general the tests consist of performing a scalar multiply for a specified runtime. These tests aim to simulate process cache access and utilization and provide repeatable results for user and kernel event counters. For example, some examples heavily utilize memory while others operate on smaller subsets of data. Additionally other processes are designed to run for longer or shorter periods of time to represent variability in task runtime.

While the actual measured runtime is presented in jiffies, these measurements should rarely be compared as they are rough measurements based on the CPU timer. Additionally, tasks which may have a variable amount of IO with the FTDI output device are likely to see unrepresentative results compared to the actual execution of tasks. Finally it is also important to note that all data fits into the L2 cache and thus pipeline stalls involved when fetching from main memory are limited the initial cold start.

To describe the behaviour for execution some terms should be defined.

- **Work**: size of data used by a **single** thread relative to a processors L1 data cache
  - uint64_t[20][25] = 25% L1 data cache = small matrix
  - uint64_t[40][50] = 100% L1 data cache = medium matrix
  - uint64_t[80][100] = 400% L1 data cache = large matrix
- **Samples**: the global number of work entries available for execution by **all** threads
  - 4 independent work entries = high spatial locality
  - 16 independent work entries = low spatial locality
- **Difficulty**: a relative measure of bounded execution time by a single thread
  - 10^2 repeated computations = easy
  - 10^3 repeated computations = normal
  - 10^4 repeated computations = hard

Figure 3: single core baseline comparison

Type	Total	User	Kernel
Instrs	467617788	466380022	1237766
Cycles	693267321	691411106	1856215
Access	264291472	263665636	625836
Refill	1539	1340	199
Runtime	45374884	-	-

(a) single core - matrix 1x utilization

Type	Total	User	Kernel
Instrs	1858506765	1853566311	4940454
Cycles	2768987855	2760678046	8309809
Access	1050488228	1047990323	2497905
Refill	1389770	1293188	96582
Runtime	178608294	-	-

(b) single core - matrix 4x utilization

The examples serve as baselines for comparing behaivour when scaling the problem to multiple cores. The first example will operate on a small matrix while the second uses a large requiring a full cache refill on

every iteration (with some degree of randomness relating to the replacement policy). On the second data set, the results are predictably much larger due to frequent data thrashing. However, despite a huge difference in APR, the total relative runtime seems to be fairly independent. This is likely a side effect of the L1 automatic prefetcher which when a pattern is detected, will start linefills in the background [5]. This prefetcher likely will cause skewed results for most of the computation since the tests have a very consistent access pattern. Thus in general, cache APR will have a higher impact especially from the schedulers perspective; where there is unlikely to be similar patterns between user processes.

Figure 4: strided matrix multiplication running against different schedulers

Type	Total	User	Kernel
Instrs	1316124843	1313462560	2662283
Cycles	1960863928	1956137419	4726509
Access	744181293	742708916	1472377
Refill	537874	509212	28662
Runtime	41175374	-	-

(a) multi core - global queue

Type	Total	User	Kernel
Instrs	1168350039	1164789764	3560275
Cycles	1734415365	1728844371	5570994
Access	660416972	658632902	1784070
Refill	69361	47662	21699
Runtime	40603738	-	-

(b) multi core - cpu queue (affinity)

Extending the previous examples to a multicore system serves as a demonstrate to the benefit of well thought out parallelization. A large matrix is separated into four medium strided matrices and will execute at hard difficulty. They will be executed against the global scheduler and the core scheduler with fixed process affinity to separate computation onto each of the four cores. Without comparing the differences between the two schedulers, they both outperform the single core computation on the same problem size. This was particularly interesting since it is not inherently obvious what may be causing this superlinear speedup.

Likely the biggest factor to this speedup is the result of data being precached in the L2 cache by the other processors. As a result of striding the computation, the innermost matrix edges will exhibit high spatial locality; while having low temporal locality. This low temporal locality effectively acts as a form of prefetching for subsequent accesses by other threads. This then results in data being present in the L2 cache well before the other processor would request access to nearby data from main memory.

TODO: Possibly mention kernel decrease in performance... this was suprising

Figure 5: 60 threads consisting of 48 hard processes and 12 easy processes

Type	Total	User	Kernel
Instrs	1543135939	1396850683	146285256
Cycles	3146576481	2076522276	1070054205
Access	845516444	789349384	56167060
Refill	7653514	112360	7541154
Runtime	152657484	-	-

(a) without pull migration (high locality)

Type	Total	User	Kernel
Instrs	1468299596	1398454567	69845029
Cycles	2552848279	2079318879	473529400
Access	816022596	790256656	25765940
Refill	3126952	140213	2986739
Runtime	156804356	-	-

(b) with pull migration (high locality)

Type	Total	User	Kernel
Instrs	1613944771	1404552990	209391781
Cycles	3491698429	2086636583	1405061846
Access	873214573	793699999	79514574
Refill	9792188	28820	9763368
Runtime	152321428	-	-

(c) without pull migration (low locality)

Type	Total	User	Kernel
Instrs	1520575475	1445743126	74832349
Cycles	2574353547	2147872630	426480917
Access	843643737	816978128	26665609
Refill	2283806	34930	2248876
Runtime	161631896	-	-

(d) with pull migration (low locality)

To test the behaviour of the system under a rapid stream of terminating processes difficulty was split into 12 hard threads and 60 easy threads designed to quickly terminate. For each example, small matrices

were tested under both cases for spatial locality.

The impact of pull migration is immediately evident in a 15-20% cycle count and 25% refill rate differentials with push migration. Despite having apparent cache benefit, the significant improvement is actually in the kernel. With pull migration, the total number of kernel instructions and cycles drops significantly. This is likely related to decreased core lock contention.

When a core quickly empties, the remaining cores will attempt to push work onto that idle queue under normal scheduling. Since all cores are competing for the idle cores lock, the cores effectively serialize themselves until they are balanced. Since a full balancing may require several context switches this affect can be quite common. However, with pull migration, instead of having many repeated requests to the idle core runqueue, a busy queue will offload work until both are balanced, hence decreasing the number of context switches and lock contention required to balance the queues.

Figure 6: even distribution of easy, normal and hard threads

Type	Total	User	Kernel
Instrs	2945961318	2829092653	116868665
Cycles	4881259967	4206238780	675021187
Access	1642115209	1598721424	43393785
Refill	4570467	298213	4272254
Runtime	175864992	-	-

(a) without pull migration

Type	Total	User	Kernel
Instrs	2727027846	2625810066	101217780
Cycles	4421745433	3903078399	518667034
Access	1520521397	1483845439	36675958
Refill	3226996	222090	3004906
Runtime	171909144	-	-

(b) with pull migration

While the previous example demonstrated the short term benefit of pull migration, this case utilizes an even distribution of difficulty operating on a small matrix with low spatial locality between threads to demonstrate long term scheduling benefits. Processes at the end of a runqueue will exhibit poor temporal locality and hence are less likely to have valid cache data by the time they run. Thus moving a process from the tail of a runqueue will end up minimizing the migration penalty of moving a recently executed process. In this case, when a group of processes dies, the cache space that they used can now be evenly distributed across fewer processes resulting in better utilization while decreasing thrashing. This decrease in cache pollution enables more work per time slice and leads to fewer context switches; thus resulting in quicker execution time.

Figure 7: comparison between thread pool and fully threaded execution

Type	Total	User	Kernel
Instrs	836416320	753346040	83070280
Cycles	1724034904	1123196400	600838504
Access	456170314	426134109	30036205
Refill	4027452	231590	3795862
Runtime	165276806	-	-

(a) threads (high locality)

Type	Total	User	Kernel
Instrs	811243405	741653152	69590253
Cycles	1789217832	1113471653	675746179
Access	444450115	419398612	25051503
Refill	4540858	379326	4161532
Runtime	66534692	-	-

(b) pooled (high locality)

Type	Total	User	Kernel
Instrs	819345284	755845609	63499675
Cycles	1672284752	1125235476	547049276
Access	450878090	427546852	23331238
Refill	3740754	148469	3592285
Runtime	167143162	-	-

(a) threads (low locality)

Type	Total	User	Kernel
Instrs	840132648	753613514	86519134
Cycles	1849770029	1127098953	722671076
Access	457128942	426166739	30962203
Refill	4435166	143616	4291550
Runtime	66158114	-	-

(b) pooled (low locality)

In Fig. 7, a work sized at 25% of the L1 data cache is distributed to 64 threads scheduled into independent

threads assigned work **or** 16 worker threads pulling work from a protected queue. They run under the same difficulty; however, locality between the two classes of samples is evaluated separately.

TODO: try this with semaphore TODO: remeasure runtimes? repeat tests with different MD case? TODO: mention that these were unexpected, was expecting much better performance with the pooled - md This example has ipc larger in both cases. However, one noticeable factor is that for larger sample size there is less l1 thrashing relevant in the APR (contributing factor for larger runtimes). Instead for the smaller case the caches are able to be fully utilized.

5 Conclusion

Conclusions here [7] [8]. **TODO: talk about the L2 cache in conclusion as it amplifies performance penalty**

At the core of most systems, an OS facilitates process execution with synchonization primitives and multiprocessing. Compared to a coopoerative OS, a preemptive OS presents additional challenges relating to scheduling and cache performance. In general since a process may be preempted at any point during execution it can be difficult to reason about performance in a local context. From a global context, a preemptive system will incur additional costs relating to the frequency of context switching and cache pollution when not dealing with shared memory. The scheduler plays an important role in determining execution order and CPU binding to minimize this overhead. In SMP systems, the general goal to balance execution across cores tends to be a fairly good approach. However, it does not address the costs related to preemption. Primarily with cache pollution, naieve balancing will tend to over-pollute cores. Thus more robust approaches involving CPU binding can help minimize this cross core pollution; yet, easily reintroduces the problem of unbalanced computation.

To maintain the benefit of CPU binding while attempting to keep computation balanced, process migration plays an important role. A short term and long term load balancer provide different benefits to global performance. The short term load balancer, ensures that work is consistently evenly distributed across cores; however, it suffers performance penalties when migrating many processes. Specifically there tends to be a high cache penalty when migrating processes with the short term scheduler. For long term load balancing, pull migration tends to migrate tasks which have a lower cache penalty.

It is important to note that for all tests, despite having a cache penalty; much was offset by the ARM prefetch module and all data fitting in the L2 data cache. This is important since other programs may not be easily classified by the prefetch module and in larger systems there is often a larger cache penalty since data is less likely to be cached in the L2 cache.

6 Lessons Learned

Since the development of the operating system was done from the ground up there were plenty of learning opportunities along the way. **TODO**

Measuring success in a bare metal environment can prove to be quite difficult; thus it became important to progressively escalate to better debugging utilities. Initially, bare minimum assembly required to toggle an LED gave limited success or failure output. The jump to FTDI was helpful in debugging simple conditions and CPU registers. Finally, for tracing complex execution; having a proper debugging protocol with JLink, OpenOCD and GDB enabled better debugging for context switching, exception handling and multicore execution.

The initial JLink setup proved to be quite tricky due to incompatability with the current version of OpenOCD, mismatched toolchain and microcontroller architecture/configuration. However, the end result was incredibly important to establishing a sucessfull development workflow. Instead of having to perform a careful SD card dance between my computer and the Pi, having the capability to flash to RAM allowed for much quicker development cycles. Initially, I had looked into alternative bootloaders (UBoot, GRUB) which

can work over ethernet, WIFI, or serial connections; however, in general the end result of having proper host debugging utilities was invaluable.

On the topic of toolchain incompatability, code was initially compiled using `arm-none-eabi` which is used for cross compiling to 32-bit ARM and while aarch64 offeres compatability with older architecture, the generid debug interface is much less compatible. Thus it is important to recognize that despite older systems often having more documentation; the complexity of developing on a compatability layer is not worth the extra effort. Additionally, the aarch64 architecture proved to have clear and better structured semantics especially related to exception handling. The exception model for aarch64 is standardized across four types; whereas, the ARMv7-A architecture had various subsets of banked registers and return instructions depedent on the interrupt source. Additionally, the addition of a unified exception status register (ESR) was much more helpful with determing the cause of a processor fault.

Newlib was designed to provide the C library on embedded systems [10] with minimal system support handled by low level functions. For example, the overhead of implementing the `malloc` and `printf` routines was reduced to simply stubbing out `sbrk` and establishing a minimal device descriptor table for `stdout`. As I built the system with the design of having POSIX compliancy, I additionally included the `<sys>` headers from Newlib; which often imposed lower level structure restrictions. For example, when implenting CPU sets I wanted to follow the same type of implementation as embedding the affinity set in the `pthread_attr_t` which was hidden around several macro guards; which when enabled led to conflicts in other areas. Thus for long term kernel development; having custom POSIX headers is beneficial to customizablility of the system and allows the system to not be constrianed by "immutable" or **static** headers.

7 Future Work

- exception higherarchy from EL0 up to EL1

- better POSIX compliancy thus port more complex programs to run tests against. eg. real programs; what would the necessary components be?, networked bounded programs, file bounded programs

- device drivers in multicore domain, network drivers would likely be interesting to explore. also sockets and their implementation

- user virtual address space and compare the performance penalty for TLB reloads

- slab allocation algorithms for a more complex system

- implementation of the linux rb-tree scheduler with implications for migration, variable quantum size + penalizing bad programs

- better migration policies with book-keeping.

- microkernel architecture + performance

- run to idle then pull, with idle sleeping

References

[1] Holdren, J. P., Lander, E., and Varmus, H. Report to the president and congress designing a digital future: Federally funded research and development in networking and information technology, 2010. https://www.cis.upenn.edu/~mkearns/papers/nitrd.pdf.

[2] Ltd, A. Arm cortex-a53 mpcore processor technical reference manual — about the l1 memory system – arm developer, 2018. https://developer.arm.com/docs/ddi0500/e/level-1-memory-system/about-the-l1-memory-system.

[3] Ltd, A. Arm cortex-a53 mpcore processor technical reference manual — about the l2 memory system – arm developer, 2018. https://developer.arm.com/docs/ddi0500/e/level-2-memory-system/about-the-l2-memory-system.

[4] Ltd, A. Arm cortex-a53 mpcore processor technical reference manual — data cache coherency – arm developer, 2018. https://developer.arm.com/docs/ddi0500/e/level-1-memory-system/cache-behavior/data-cache-coherency.

[5] Ltd, A. Arm cortex-a53 mpcore processor technical reference manual — data prefetching and monitoring – arm developer, 2018. https://developer.arm.com/docs/ddi0500/e/level-1-memory-system/data-prefetching/data-prefetching-and-monitoring.

[6] McCool, M., Robison, A., and Reinders, J. *Structured parallel programming: patterns for efficient computation*. Morgan Kaufmann, 2012.

[7] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2015.

[8] Schimmel, C. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, 1994.

[9] Van loo, G. Arm quad a7 core, 2014. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf.

[10] Vinschen, C., and Johnston, J. Newlib, 2016. https://sourceware.org/newlib/.