Analysis of SMP Kernel Scheduling

Jnaneshwar Weibel
University of British Columbia

April 23, 2018

Abstract

Algorithms have become more distributed and more resource intensive; thus, requiring computer hardware to become increasingly efficient to effectively run modern 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, with respect to load balancing and locality of access to shared memory, when distributing programs that can influence the performance in both 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 is applicable 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 determining an execution order for user threads. 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 determine their impact on system performance.

The performance of several tasks will be measured using a single global queue, a per-core ready queue without pull migration, and a per-core ready queue with pull migration. Pull migration refers to periodically re-balancing the load on two queues. In addition, some tasks will be constrained to a smaller set of CPUs and; thus, may be fixed to a single processor and undergo migration. This paper will demonstrate the performance implications that various scheduling algorithms have on both local and global task execution.

2 Environment

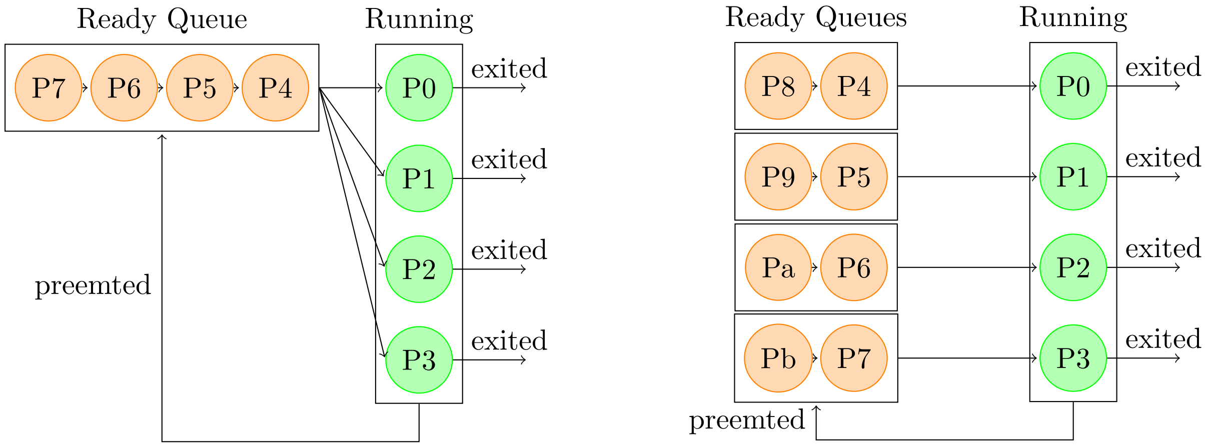
The OS will be compiled for the ARMv8 (aarch64) instruction set and will be executed on the Raspberry Pi 3 B. The device includes a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor where each core has a 16KB L1 data cache, a 16KB L1 instruction cache, and a shared 128KB L2 cache [2] [3]. The caches operate using the MOESI protocol with intra-core cache coherency and caches 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 every 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 Design

A ready queue can be defined as a generic data structure with two primary operations: ready and next, and in general will consist of a FIFO list of tasks. Internal queue manipulation is protected by a spinlock; however, a queue with multiple priorities will have a spinlock for each priority queue minimizing contention when scheduling with different priorities. However, for this discussion we will assume that all (non-idle) processes run at the same priority.

A naive implementation for a SMP scheduler involves a global ready queue which all cores pull processes from. This implementation acts identical to its implementation in a single core system, but requires explicit synchronization. On SMP systems this data structure is beneficial since it guarantees equal workload distribution across all cores in addition to implementation simplicity. Additionally, this distribution is not affected by a rapid stream of exiting processes. However, it 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, 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. 2).

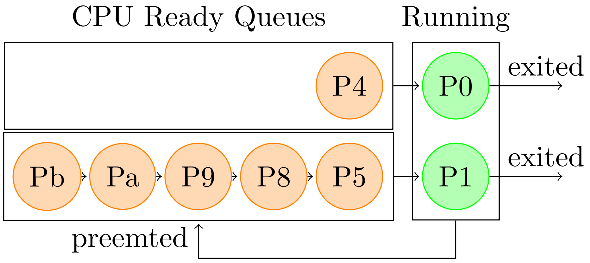


Figure 2: Unbalanced Scheduling

Pull migration is an effort to improve the re-balancing latency for queues which exhibit the behavior in fig. 2. 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 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.

With respect to fairness, there are several areas which should be discussed. With the global queue, all processes are allocated the same time slice and have no core preference to help distribute the cache load. Effectively, this means that all process share the same level of execution fairness; however, with per-core runqueues, core queue imbalance leads to execution latency on busy queues compared to that of an idle queue. Migration aims to reduce the latency due to imbalance; yet it introduces migration unfairness. For example, migrating a process will lead to a full L1 cache reload on the new core. Thus, the initial form of migration during preemption tends to be relatively unfair towards recently executed processes, because they are the most affected by a cache reload. Since processes are migrated from the end of the queue, a process is guaranteed to execute sooner on the idle queue. Additionally, processes at the end of a queue exhibit poor temporal locality relative to the rest of the queue; and hence the cache penalty on migration is minimized as a partial reload may be required by the time the process is scheduled.

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.

Total runtime is also presented in clock ticks; yet, is generally inaccurate since it includes the I/O overhead when dumping metrics over `stdout`. Additionally, all data comfortably fits into the L2 cache and; thus, pipeline stalls involved when fetching from main memory are limited to the initial cold start.

To describe the behavior 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

4.1 Trivial Multiprocessing

Fig. 3 serves as a baseline for comparing behaviour when scaling the problem to multiple cores. The example operates on a small matrix, while the second uses a large matrix requiring a full cache refill on each 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

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 - medium matrix

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 - large matrix

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 demonstrates 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 both the global scheduler and the core scheduler with fixed process affinity to separate computation onto each of the four cores (fig. 4). Without comparing the differences between the two schedulers, they both outperform the single core computation on the same problem size. Surprisingly, both exhibit 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 innermost matrix edges will exhibit high spatial locality; while having low temporal locality. This temporal locality effectively acts as a form of prefetching for subsequent accesses by other threads to the L2 cache.

4.2 Pull Migration

Figure 5: threads with high spatial locality with a subset that quickly terminates

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

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

To test the behavior 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. Small matrices were tested with high

Figure 6: threads with low spatial locality with a subset that quickly terminates

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

(a) without pull migration

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

(b) with pull migration

(fig. 5) and low (fig. 6) spatial locality.

The impact of pull migration is immediately evident in a 19-26% cycle count and 60-76% refill rate decrease with push migration. Despite having apparent cache benefit, the significant improvement is 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 7: 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 (fig. 7) 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.

4.3 Thread Pool

In fig. 8, 64 threads operating on a small matrix are 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.

For these examples, the primary difference is evident in the thrashing in the kernel. As more processes become preempted, the total amount of active memory increases leading to full cache utilization by user processes. In general, this leads to eviction of kernel cache lines; resulting in a 22-31% increase in kernel IPC. However, pooling does result in poorer performance with regards to shared memory. Since the active

Figure 8: thread pool vs individually threaded execution with varying spatial locality

Type	Total	User	Kernel
Instrs	858992530	753437017	105555513
Cycles	1910983635	1123087469	787896166
Access	465293588	426171724	39121864
Refill	5513845	210562	5303283
Runtime	165487634	-	-

(a) individual threads (high locality)

Type	Total	User	Kernel
Instrs	825909448	773161950	52747498
Cycles	1462457892	1162415850	300042042
Access	453475978	435856372	17619606
Refill	1737886	364441	1373445
Runtime	61557018	-	-

(b) pooled (high locality)

Type	Total	User	Kernel
Instrs	850652290	757440549	93211741
Cycles	1900834361	1127333154	773501207
Access	463377065	428435448	34941617
Refill	5282763	128873	5153890
Runtime	165916644	-	-

(c) individual threads (low locality)

Type	Total	User	Kernel
Instrs	819945002	764500953	55444049
Cycles	1522878029	1144994038	377883991
Access	450320681	431461874	18858807
Refill	2026635	131657	1894978
Runtime	61636292	-	-

(d) pooled (low locality)

subset of work is much more constrained, two processes which may be working on the same subset of data are unlikely to be scheduled on the same core. When there is relatively low locality between workers, the difference between single threaded and pooled is minimal; however, with high locality, pooling results in a 41% reduction in APR. For kernel performance, the largest factor is evident in the reduction in kernel instructions executed; this is a result of avoiding the synchronization and computation overhead when spawning a new process and instead reusing existing threads. Avoiding this overhead results in 40-50% reduction in kernel instructions. It is also important to note that the reduction in user performance is more apparent when processes exhibit high spatial locality as processes will share much of the cache. Despite user performance slightly decreasing due to poorer cache utilization, there is significant improvement in kernel execution that results in a net increase in runtime.

5 Conclusion

At the core of most systems, an OS facilitates process execution with synchronization primitives and multiprocessing. Compared to a cooperative 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 good approach. However, it does not address the costs related to preemption. Primarily naive balancing will tend to over-pollute cores; thus, more robust approaches involving CPU binding can help minimize this. Yet, CPU affinity easily reintroduces the problem of unbalanced processors.

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. While, this load balancing does introduce more lock contention between the cores, “contention on these locks is rare unless the workload exhibits grossly overactive context switching and thread migration” [7]. In theory, this approach should scale well to systems with many more processors as each core is relatively independent.

Most of the examples were generalized to a consistent access pattern used to simulate application behavior. "Trying to judge cache performance with benchmarks and generalizing the results must be done with care" [8], as much of these patterns are for very specific examples and do not necessarily scale. Nevertheless, some generalizations can be made with respect to application development. Separating computation of completely unrelated data to other processors using constrained affinity will generally result in better performance than leaving scheduling entirely to the OS. This also applies to moving tasks where the data is disjoint yet has some spatial locality, as disjoint affinity can enable prefetching into the larger levels of the cache hierarchy. From the global scheduling perspective, it would be useful to stride data to have less cache impact during preemption; yet, an application developer will in general want to consume as much memory up to cache boundaries to achieve the best local performance. Finally, avoiding the additional process creation and destruction overhead using thread pools should often be considered for lightweight and cache intensive processes.

It is important to note that despite some tests having a large cache penalty; much was offset by the ARM prefetch module and all data fitting into the L2 data cache. In general, across a system, programs will not display the same memory access patterns that are easily detected as the test programs. Additionally, the cache penalty would tend to be much higher when larger amounts of data is accessed that cannot comfortably fit in the L2 cache. The Pi 3 offered a very symmetric memory and core layout; however, other core-memory relationships can include "CPUs sharing a layer of cache, CPUs that are local to a particular memory, or CPUs that share execution units such as in a system with symmetric multi-threading" [7] and should be considered as the most important area for fine-tuning performance for a specific system.

6 Lessons Learned

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 incompatibility with the current version of OpenOCD, mismatched toolchain and microcontroller architecture/configuration. However, the end result was incredibly important to establishing a successful 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 or serial connections; however, in general the end result of having proper host debugging utilities was invaluable. Having had debugging utilities from the beginning, compiling with semihosting support `--specs=rdimon.specs -lc -lrdimon` might have been a good alternative to have POSIX compliancy and to get up and running quicker.

On the topic of toolchain incompatibility, code was initially compiled using `arm-none-eabi` which is used for cross compiling to 32-bit ARMv7-A and while ARMv8-A offers compatibility with older architecture, the generic 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 compatibility 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 dependent on the interrupt source. Additionally, the addition of a unified exception status register (ESR) was much more helpful with determining 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 to redirect stdout to the serial output. 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 implementing 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 customizability of the system and allows the system to not be constrained by “immutable” headers.

7 Future Work

Initially, some easier goals that do not necessarily involve much design modification include, migrating user execution to EL0 and implementing user virtual address space. This may adds additional overhead on context switching relating to reloading the TLB and possible cache invalidation on MMU-less architectures that would be interesting to investigate. Additionally, adding exception handlers for EL2 would be useful for adding double fault handlers and warm-reset functionality for smoother development workflows.

From the metrics gathered from this project, process migration is an area that could benefit from improvements. For example, as the kernel benchmarks process execution, the scheduler could adapt to determine an execution order which minimizes thrashing. Additionally, moving to a scheduler system such as Linux CFS based on red-black trees and and variable quantum size as a method of adjusting execution fairness. In particular, fairness may be able to be represented as a function of cache performance and hence the quantum could be scaled in order to achieve better global performance. Additionally, as kernel complexity is added, it would be useful to look into kernel cache optimization including applications of slab allocation.

Finally, some topics which will require large amounts of structural refactoring include implementing communication routines to implement a microkernel or comparing execution to asymmetric multiprocessing. For asymmetric multiprocessing, instead of focusing on throughput; comparisons in power consumption utilizing the ARM WFE/WFI + SEV instructions; which enable the processor to be placed into a very low power sleep mode. This could be used in conjunction with a master-slave kernel which preserves the uniprocessor execution environment from the kernel's point of view and thus can run on an SMP system without race conditions [8]. A master-slave processor would could allow most cores to sleep until the master detects work. In addition, the master may chose to disable cores when there is minimal work. Since locking primitives are also not required, lock contention in a spin-locked kernel could be compared to bottleknecks in a master-processor. In all a kernel is a very complex system that offers boundless opportunities for optimization and experimentation.

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/`.