Analysis of SMP Kernel Scheduling

Jnaneshwar Weibel

Abstract

Over the years computer hardware has grown increasingly efficient; 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 about this distribution that may influence the performance of user programs. Shared memory and locality of data ...

1 Introduction

Modern computer hardware has grown increasingly efficient to improve the performance of various systems by improving areas such as processor frequency and memory latency. While instruction throughput can be improved with optimizations to a CPU pipeline, multiprocessing is also increasingly important for improving throughput. To improve shared memory latency, processors have adopted smaller memory subsystems, or caches, that allows quicker data access.

As the hardware evolves, software must be able to take advantage of these changes. Within an operating system, the scheduler is responsible for running user tasks and determining an execution order; however, with multiple processors these tasks can be distributed across cores. In a preemptive system, a task may migrate between different processors and this migration will generally impact cache performance; however, globally this migration my improve performance. Thus several distinct policy decisions relating to task scheduling will be compared.

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 and consists of a 16KB L1 cache and a 128KB L2 cache. Intra-core cache coherency is enabled and uses the MOESI protocol [1]. 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 and each core maintains its own core timer which will preempt the current task each MS. [4]

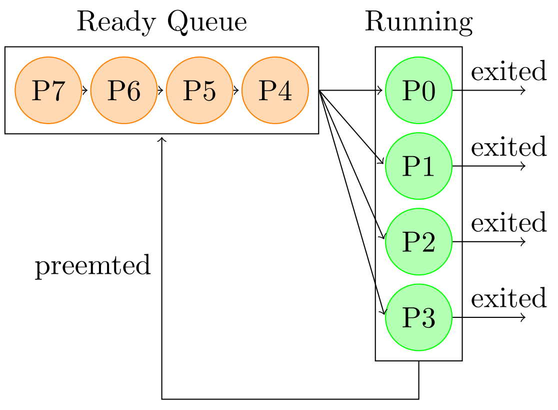

Figure 1: Global Ready Queue

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. (TODO possibly example) However, the approach suffers from kernel lock contention since all cores must synchronize access to the queue.

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

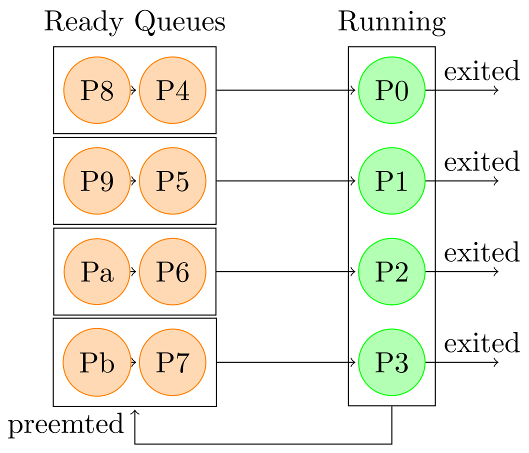

Figure 2: Per Core Ready Queue

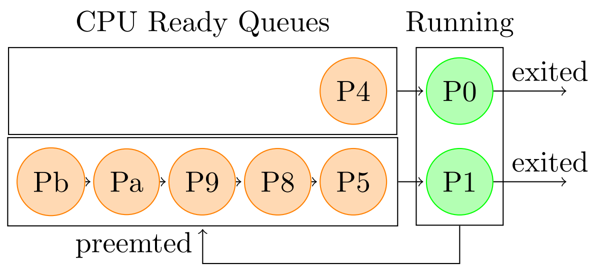

Figure 3: Unbalanced Scheduling

terminate leading to a load imbalance between ready queues (fig. 3).

Pull migration is an effort to improve the rebalancing latency for queues which exhibit the behaivour 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 rebalancing 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.

4 Performance Tests

Various user programs were tested against different variations of the scheduler. In general the tests were performed on processes tasked with performing a scalar multiply on a data set with a size relative to the L1 cache. As a baseline, the OS was configured to run on only a single core and a matrix multiplication was performed on a matrix exactly the size of the cache. For more interesting results, a data set 4x the size of the cache was tested with the global run-queue and the per-core run-queue with evenly distributed CPU affinity. In these cases each process would work on an independent part of the data set.

The impact of per-core scheduler with and without pull migration was analyzed with tasks with varying runtime. Three classes of processes were defined as short, medium and long; each, with 10x the runtime of the previous case. An even distribution of each type of task and the data set was tested against these schedulers. Additionally the schedulers were tested against many short runtime processes compared to the few long runtime processes.

Finally, a common pattern; in resource intensive and parallelizable tasks, is to create a smaller subset of OS tasks and pull work off a shared queue. Otherwise known as a thread pool, the difference between allowing the OS to manage all the tasks compared to semi-manual work queueing was examined. Work was divided into 64 threads with 16 workers for the pooled case. Additionally the locality of the data being worked on was constrained to both 4 (sm) and 16 (lg) matrices of a smaller size than the L1 cache. All tests ran on the per-core scheduler with pull migration.

In general for the following examples the runtime gives a rough measurement of elapsed jiffies. However, the cycle count provides a better peforamnce measurement. Additionally, it is important to all data is cached in the L2 cache and thus stalls for fetching from main memory are generally minimized.

This initial example aims to show the stark difference between process cache utilization. The two examples operate on a matrix fully utilizes

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

Table 1: 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	-	-

Table 2: single core - matrix 4x utilization

the L1 cache and overloads the cache in the second instance. It is important to note that while the cycle count is nearly 4x larger as expected, the refill rate for the larger matrix is larger by a factor of nearly 900x. In these examples the most noticeable difference is the instructions per cycle in the kernel being roughly 11% slower, due to the frequent thrashing of kernel memory.

Type	Total	User	Kernel
Instrs	354249559	353629318	620241
Cycles	527538552	526403550	1135002
Access	200305644	199963039	342605
Refill	125708	116722	8986
Runtime	41175374	-	-

Table 3: multi core - global queue

Type	Total	User	Kernel
Instrs	375176881	374755150	421731
Cycles	556693572	556049701	643871
Access	212114934	211903384	211550
Refill	15332	14735	597
Runtime	40603738	-	-

Table 4: multi core - cpu queue (affinity)

Extending the trivial example to multicore serves as a demonstration of easily parallelizable code and the benefit of process affinity. For this example, the instructions per cycle is nearly 1%

faster in the kernel for the case with constrained affinity. Again, the kernel data structures generally remain in the L1 cache as evident by the a lower number of access per refill $\approx 1.6\%$ in the global runqueue case. It is also important to note that the number of kernel instructions is $\approx 68\%$ less in the per core runqueue, howing to less lock contention between cores.

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

Table 5: death - no pull sm

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

Table 6: death - pulled sm

Notes: lower cache locality leads to fewer lost cycles, additionally migration without pull migration tends to lead to more kernel contention. Likely because full cores try to migrate to the local core...

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

Table 7: death - no-pull

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

Table 8: death - pulled

The previous examples operate on TODO. TODO: this case needs fleshing out... specific pattern likely samples of the same matrix get flushed to the idle cores. May want to repeat these tests with affinity for the idle procs.

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

Table 9: runtime - no pull

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

Table 10: runtime - pulled

Same characteristics as death example

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

Table 11: pool - unpooled sm

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

Table 12: pool - pooled sm

This example has ipc larger in both cases. However, one noticeable factor is that for larger sample size there is less l1 thrashing relavent in the APR (contributing factor for larger runtimes). Instead for the smaller case the caches are able to be fully utilized.

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

Table 13: pool - unpooled

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

Table 14: pool - pooled

5 Conclusion

Conclusions here [2] [3].

References

[1] 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`.

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

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

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