Operating System and Applications Multi-Core Scalability Issues

Edward Hills

***Abstract*—Multi-core scalability is in the fore-front of every programmers mind when programming operating systems or any serious application programs. For a number of years now it has been impossible to increase a single cores clock speed due to several limitations, mainly heat and power consumption. This is why a new paradigm has arisen in which we put more and more cores into a single CPU. The next decade is expected to herald 100s or 1000s of cores on a single chip. This is why we must begin to start thinking about how we can scale our operating systems and applications to gain the full benefit of multi-core. This paper will talk about some methods to attack scalability as well as propose some new ideas.**

I. Introduction

SCALABILITY is an important issue in todays modern operating system era. Current operating systems (OSs) are being retro-fitted with techniques to increase their scalability. On a whole things are improving, and this can be seen in the versions of the Linux kernel which are being released. However, some researchers see this is a stop-gap rather than a solution. This is due to a number of factors which cannot simply be fixed to be more scalable due to their semantics. Some scalability issues include:

- Global or coarse-grained locks
- TLBs
- Shared Memory
- Semantic Serialization
- Cache misses
- Unnecessary Resource Sharing

Coarse-grained locks are a major scalability issue in multi-core or even highly threaded systems. A coarse-grained lock is one that locks a larger area than it possibly needs to to accomplish a job that needs to run without interference or the possibility of a race condition. Original linux kernels had whole kernel global locks but these were quickly removed with smaller finer-grained locks, however these locks are still not fine enough in a multi-core environment.

TLB or Translation Look-aside Buffer holds a table of physical addresses as generated from the virtual addresses, these are needed to be global so that each core gets the same data, when updating or viewing the TLB a look must be held which stops all other processes from accessing it, for such a widely used data structure this can severely limit scalability.

Shared Memory is simply a region of memory that is shared amongst different cores, this can cause a range of problems which are discussed later.

Semantic serialization is the problem in which some code blocks must be run in serial just to the nature of their task, these are one of the hardest problems to overcome and the semantics need to be rethought and designed to remove these.

Having one core read what another core just wrote imposes a wide range of cache misses, if the same core that did the writing did the reading than it would be able to access it from cache and save on roughly 200 cycles to get it from memory.

Unnecessary resource sharing often occurs as side effects of bad programming or lack of thinking about scalability. Some resources simply do not need to be shared and we will see examples of this later.

Throwing more cores or processors at a problem may not be the ultimate solution however, as Amdahls Law [2] shows, we are limited by the sections which must be run in serial, if a program spends 1 hour out of 20 (95%) executing serial code then Amdahls Law shows that we can only speed up computation by 20x. This is why serial parts must be rethought and redesigned to be parallelised.

In this paper I will talk about previous papers and the techniques which have helped overcome these problems such as *sloppy counters* [3], use of *Read-Copy-Update* [5] techniques and OSs which have done a full re-think of the kernel design with scalability in mind such as *fos* [6].

** talk about your idea when you have one here **

II. Related Work

Multi-core CPUs have been around for over a decade now and with this plenty of time to adjust and come up with ideas to help improve the scalability issues that we face. This section will discuss some issues and previous solutions to overcome these, as well as some new designs or ways of thinking about the issue of scalability.

III. Current OS Solutions

Without changing drastically the way we think about modern operating systems we have no choice but to examine areas of the kernel we have now and find ways to limit the amount, what, where and when we share resources among each thread or core. By reducing the amount of sharing we have to do and by keeping everything as modular as we can, we can try and curb the amount of differing cores that must access the same thing. By doing this, we can improve scalability drastically.

A. Sloppy Counters

Sloppy counters proposed by Byord-Wickizer *et al.* is a way of doing *lazy updates*. The linux kernel uses shared counters for a range of tasks such managing resources and garbage collection, this a scalability issue if many cores are accessing an updating these counters. *Sloppy counters* aims to remove

this bottleneck by having each core have its own counter and update that instead of the shared counter. It does this in the hopes that it can keep spare references that new threads can use. It will have to reconcile with the central counter if the local count grows above a threshold value or when deciding whether an object should be deallocated, thus it is best used when objects are rarely deallocated.

Sloppy counters has the added benefit of being backwards compatible with other kernel counters meaning that not all sections of the kernel need to be changed by only that which imposes scalability issues. However one down side is that that they use space proportional to the number of cores present.

By adding sloppy counters to keep track of *dentrys*, *vfs-mounts*, *dst_entrys* and to keep track of memory allocated by TCP and UDP network protocols.

One downfall of this paper [3] is that it does not give specific quantifiable results when using just sloppy counters, this is because the paper benchmarks a range of applications with more than one improvement made to it.

B. Read-Copy-Update

Read-Copy-Update provides a solution to many scalability issues, an important one being contention in the address space design [4]. Clements *et al.* re-designed the address space to use a balanced binary tree, *Bonsai*, to ensure non-destructive updates of the table.

I will now briefly describe RCU and how it helps with scalability in a multi-core environment.

The main idea of RCU is that when a reader process or thread comes to read a particular block thats inside an RCU synchronization block it first copies the data that is held within the data-structure and then makes a copy of the original pointer itself. Next, it simply carries on and if need be it updates its copy. It then updates the global pointer and waits a certain grace period. It has this grace period so that all other readers which are reading the original data-structure have time to finish.

Once the code enters a 'quiescent state' (an area in the code in which you can guarantee that all previous operations have completed) then the grace period ends and the resource is reclaimed. There are several things that defines what a 'quiescent state' is, some methods simply use a counter and increment every time an operation is begun and decrement it when an operation is finished, then when the counter reaches zero it has completed everything before it. For non-preemptive systems if a context switch is performed then that could be classed as a quiescent state also. In some systems if an interrupt is called or a trap has been executed then these are also accepted.

RCU in its standard implementation is mainly for read-mostly data structures such as a routing table which is more often read than updated. One major advantage to RCU is its low-cost and low-overhead compared to normal synchronization techniques (such as a spinlock) which is relatively quite expensive and multi-processor unfriendly.

RCU is described as a 'two-phase' locking mechanism, the first phase is to carry out enough of each up-date for new operations to see the new state, but still let old operations carry on and the second phase is to finish the update once the grace period has finished.

RCU is a severe advantage over traditional locking mechanisms such as spinlocks as they are severely limited by worst-case memory latency. This is because once a traditional lock has been obtained it must write to the locked data-structure, this means that it must hit memory which is slow compared to cpu or cache.

There is a limitation however due to the *wait_for_rcu()* function not working in a pre-emptible kernel unless pre-emption is specifically disabled for that section. When implementing FD management with the use of RCU, Mckenney *et al.* [5] found that the kernel exhibited over 30% more throughput for 4 cores and even with a uniprocessor there was a slight increase in performance (0.65%).

Multi-core oriented OSs such as K42 use RCU pervasively as an existence lock. It is commonly used in Linux kernels since 2.5.

Read-Copy-Update locking technique allows multiple threads and cores to access the same data-structures without having to attain a lock which would limit scalability. Now knowing about RCU we can explain how this was used to help increase scalability in a concurrent address space.

By applying RCU to a balanced tree [4], *Bonsai* allows us to perform read operations in parallel with writes and avoid cache coherence traffic caused by read locks. *Bonsai* avoids the race conditions of keeping the red-black tree balanced by its design.

By implementing *Bonsai* into the linux kernel, applications such as *Metis* and *Dedup* can achieve near-perfect speed-up to 80 cores and the results of Clements *et al.* showed that the scalability improvement was likely to keep on being beneficial with even more cores.

IV. Future OS Solutions

Techniques like the ones I have described above can almost be seen as 'stop-gaps' to the scalability issues we face in modern operating systems. These are becoming more and more effective as we devise new techniques for increasing scalability, however there are some that believe a total rethink and redesign of operating systems needs to be done. Below I will talk about a couple of these.

A. Factored Operating-System

fos or Factored Operating System [6], is an operating system that has been designed with scalability at the forefront of the developers mind. Fos aims to be able to support future massively multi-core chips (1000+ cores) that are likely to be around in the next decade or so. It has done away with contemporary modern operating systems such as Linux and has instead been rethought and redesigned to work a lot like Internet servers do today.

Fos takes its main fundamental idea from common Internet servers that are used everyday for the web. These servers are highly scalable and can deal with millions of users at a time. In a nutshell fos has a few separate and important ideas:

- Kernel and Application code execute on separate cores
- Each core performs a set of separate services
- Main communication between cores is via message passing
- Sets of services and applications can be broken up into multiple *fleets*
- Timesharing system replaced with spatial sharing

By designing fos with the above view we instantly avoid the need for a majority of locks as well as avoided having the application and kernel interfere with each other. Below are just some of the benefits offered such as:

- Inherently avoids contention between application and kernel
- Only one thread for each core avoids locks
- Quicker access to servives and fully parellised as tasks are spatially-shared not time-shared
- Avoids expensive context switches

The system service servers execute on top of a microkernel inspired by designs such as Mach [1] however has multiple differencnes such as fos distributing and parallelizing within a single system service server and having a spatially aware placement engine/scheduler.
Fos as you can see above is a complicated and new idea which aims to make scalability with 1000+ cores on a single chip possible

B. K42

K42 blah

V. New Proposal

A. Basic Idea

My new idea to help improve scalability is to take a similar approach to that of *fos* [6] and completely rethink the idea and design of the kernel.

B. Implementation

To implement the design I have discussed above I first though to blah blah blah

VI. Conclusion

The conclusion goes here. The conclusion goes here. The conclusion goes here. The conclusion goes here.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *In Proceedings of the USENIX Summer Conference*, pages 93–112, 1986.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[4] A. T. Clements, F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

[5] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[6] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.