Improving Kernel Scalability by Lock-Aware Thread Migration

Yan Cui (graduate student) Yu Chen Advisor: Yuanchun Shi (Tsinghua University)

Multi-core architecture has been adopted by the chip manufacturers. Computers with two or four cores are common. Trends seen from Intel's 80-core [1] processor suggest that cores in a single chip are still increasing. Within ten years, each chip will embrace thousands of cores.

Commodity operating systems, however, cannot scale well with the number of cores because the lock protecting the shared data structure in the kernel tends to be time-consuming when the lock contenders increase. This phenomenon can be observed easily on the future many cores even if only several instructions exist in the critical section. In the past, operating system developers exploit finer grain locks to remove this bottleneck and this method has been proven to be successful. However, as the cores increase exponentially, making the critical section finer tends to be hard and error prone [2].

Different from previous work, we propose to improve the parallel scalability of operating systems by lock-aware thread migration. The idea is inspired by the insight that when the number of cores contend for lock is small, each core can acquire the lock quickly. Therefore, if lock contention degrades the performance and scalability, we limit the number of cores on which the lock code can execute. In our system, the limitation is achieved by migrating threads between cores. Lock-aware thread migration has three advantages at least: 1. For each thread, the time of waiting for lock can be shorten. 2. The shared data in the critical section would be fetched in less caches. This may reduce the cache misses introduced by the cache coherence protocol. 3. We implement this method by pure software. Thus, evaluation can be preformed on the real multi-core system.

We demonstrate the potential of lock-aware thread migration by setting the processor affinity. *single_ctr*,*multiple_ctr* and *doubly_linked_list* are three micro-benchmarks selected. All experiments are carried out on the same AMD Opteron 32-core machine with Linux 2.6.29.4 as the operating system. Experimental results indicate that all benchmarks with core affinity scales much better than the original system.

However, in order for lock-aware thread migration to achieve the potential, several challenges should be addressed. The first problem is that which cores should be used to execute the lock code when the scalability degradation is detected. Assign the cores statically or dynamically for lock and critical section code is a way to solve this problem. However, dynamic assignment has the advantage over static assignment in that the number of cores executing lock code can change according to the program's synchronization behaviour. Therefore, the dynamic method is adopted in our system. Specifically, core 0 is the only initialized core which is used to execute lock code. When the dynamic policy decides to utilize more cores for lock, cores with ascending core id will be added. This method takes the shared last level cache on the multi-core architecture in consideration.

Another challenge is the overhead of thread migration. On the AMD 32-core platform, migrating a thread between two cores takes 9 microseconds at least. The experimental results on the *single_ctr* micro-benchmark indicate that the time of completing 100,000 operations using 32 threads increases to 33 seconds when migrating threads before and after each critical section (the time is 12.58 seconds with the original Linux). For this challenge, we propose to migrate threads according to the hotness of each lock. If the lock hotness is larger than a threshold, threads waiting for this lock should be migrated. Meanwhile, to adapt the program's dynamic synchronization behaviour, lock contenders on the lock cores should be migrated back when the lock hotness is smaller than a threshold.

The third challenge is the cores executing the lock code may become a bottleneck. When one or more locks are highly contended, too many threads are migrated to the lock cores. The runqueues of the lock cores become so long that each thread cannot be scheduled frequently enough. We propose to solve this problem by detecting the average load of the lock cores and increasing the number of lock cores properly. Using the technologies discussed above, we expect that the lock-ware thread migration can effectively improve the kernel scalability on the multi-cores for a wide range of workloads.

References

[1] VANGAL, S., HOWARD, J., RUHL, G., DIGHE, S., WILSON, H., TSCHANZ, J., FINAN, D., IYER, P., SINGH, A., JACOB, T., JAIN, S., VENKATARAMAN, S., HOSKOTE, Y., AND BORKAR, N. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *Proceedings of the IEEE International Solid-State Circuits Conference* (2007), pp. 98–99.

[2] WENTZLAFF, D., AND AGARWAL, A. The case for a factored operating system (fos). In *MIT-CSAIL-TR-2008-060* (2008).