Brief Announcement : Persistent Unfairness Arising From Cache Residency Imbalance

Dave Dice
Oracle Labs
dave.dice@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Nir Shavit
MIT
shanir@csail.mit.edu

ABSTRACT

We describe a counter-intuitive performance phenomena relevant to concurrency research. On a modern multicore system with a shared last-level cache, a set of concurrently running identical threads that loop – each accessing the same quantity of distinct thread-private data – can suffer significant relative progress imbalance. If one thread, or a small subset of the threads, manages to transiently enjoy higher cache residency than the other threads, that thread will tend to iterate faster and keep more of its data resident, thus increasing the odds that it will continue to run faster. This emergent behavior tends to be stable over surprisingly long periods.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Performance, experiments, algorithms

Keywords

Concurrency, threads, caches, multicore

1. INTRODUCTION

Multiple threads concurrently accessing a shared last-level cache (LLC) can encounter competition for cache residency and *destructive interference*. This behavior is well-known [2]. We observe, however, that such a group of homogeneous threads accessing private data regions can suffer from significant imbalance in their relative rates of progress. We found this behavior somewhat surprising and a possible confounding factor for experiments and benchmarks.

After analysis, we determined this unfairness occurs when a small subset of the threads manage to achieve relatively better LLC residency. Those threads then tend to run faster, which in turn supports their occupancy. This state can be stable over periods of minutes. When this form of residency-based unfairness is present, the progress rates of the threads tend to be bimodal : threads are either *fast* or *slow*. A thread's LLC miss rate and observed cycles-per-instruction (CPI) is inversely proportional to – and inversely correlated with – that thread's progress rate. The onset of the effect occurs when the sum of the private working sets of all the threads is near the capacity of the shared LLC. The stratification effect abates when the working set grows sufficiently larger than the cache capacity and most accesses miss in the LLC.

To further establish etiology and causation, we performed an experiment where we intentionally inserted a transient stall, and were able to transform a fast thread to a slow thread, as the stall caused the thread to lose LLC residency. Once disadvantaged, the thread remained slow.

We believe this new phenomena should be noted by designers and researchers and added to the set of existing concerns – such as false sharing, for example – taken into account when analyzing concurrent program behavior.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
ACM 978-1-4503-2821-0/14/06.
http://dx.doi.org/10.1145/2612669.2612703.

2. EVALUATION

Figure 1 depicts the magnitude of the phenomena on an Oracle SPARC T4-1 [6] system running Solaris 11 and an x86 system running Linux 3.11 on an Intel i7-4770 "haswell" processor. The T4-1 has 8 cores with 8 pipelines per core for a total of 64 logical thread contexts. The shared L3 LLC is 4MB with a non-most-recently-used (NMRU) replacement policy and is inclusive of the L1 and L2 caches. The i7-4770 has an 8MB LLC shared over 4 cores with 2 pipelines per core for a total of 8 logical thread contexts. Both systems are single-socket non-NUMA and have 64-byte LLC cache lines, and both have core-private L1 and L2 caches.

Each thread allocates a circular ring of intrusively linked nodes which form the thread-private data. Each node resides on its own cache line and is 64-bytes in length. The order of the nodes is randomized in order to reduce the influence of automatic hardware prefetch facilities. To reduce the influence of TLB pressure, each thread invokes mmap() to allocate a set of contiguous pages from which its set of nodes is in turn allocated via a simple "bump pointer". This gives the system the opportunity to provision the region underlying the nodes with large pages, and minimizes the number of pages underlying a given ring. This approach also ensures balanced placement of nodes over cache indices [1]. The number of elements in the ring – the circumference – is configurable via command line parameters.

We use a 60-second measurement interval during which each thread executes a top-level loop that repeatedly traverses its ring. At the end of the measurement interval the benchmark reports the number of traversals completed by each of the threads.

In our experiments we ran with 8 threads for all data points. Both the Linux and Solaris schedulers are work-conserving and all

8 threads ran for the entirety of the measurement interval. With just 8 ready threads, Solaris will place each thread on its own core in order to avoid contention for pipeline and L1/L2 resources. That is, Solaris balances the distribution of threads over core-level resources. On the Linux x86 platform, pairs of threads share a core.

On the X-axis we show the total size of all thread-private data expressed as a fraction of the LLC size. On the Y-axis we report the *spread* – the degree of unfairness as defined by the number of traversals completed by the fastest thread divided by the number of traversals completed by the slowest thread. We report the median spread from 7 runs. For our experiments *turbo boost* was disabled on the Intel processor and the clock rate was fixed at 3.4GHz for all cores.

In other experiments we replaced the per-thread rings with fetches from randomly selected locations in thread-private arrays. This variant exhibited similar unfairness.

In supplemental experiments we modified the loops executed by the threads to first acquire a contended central MCS lock [5], followed by by 10000 steps of a register-based random number generator [4]. No memory is accessed in the critical section. The thread then releases the lock and traverses its ring. No store instructions are executed, so there is no induced coherence traffic. We opted for a top-level loop with an MCS lock and critical section to illustrate that the problem manifests even under a fair FIFO lock. As configured, the lock is contended and threads typically wait for entry. Again, we observed similar levels of unfairness. It is commonly the case that a slow thread S might release the lock, passing ownership to some fast thread F, only to find that F completes the critical and non-critical sections more quickly than S can complete its non-critical section, so F races ahead and queues on the lock before S. F can erode S's residency faster than S can erode F's residency, so F maintains its relative advantage.

We note that being fast or slow seems to be happenstance from the perspective of the programmer, and is not to related to thread placement within the system topology or geography.

As can be seen in the graph, the onset of unfair progress rates starts near the LLC capacity, and the worst-case magnitude is over 5X for both platforms. Note that the onset occurs before the fraction reaches 1.0 because our caches are not ideal fully-associative. We believe the slightly different shapes exhibited by the T4-1 and the i7-4770 are due to differing cache architectures, the ability to leverage memory-level parallelism, and the depth of the speculation windows. Finally, notice that the i7-4770 exhibits minor unfairness when the LLC fraction is very low. We have 2 threads sharing the per-core L1 and L2 caches, and find the unfairness phenomena manifests at those levels of the cache hierarchy.

3. CONCLUSION

Cache behavior is taking an increasingly important role in multicore software design, and properly understanding cache sharing and eviction policies is often key to delivering good concurrent performance. We presented an interesting new cache phenomena whose effects should be noted by designers and researchers when analyzing concurrent program behavior.

We note that improved hardware cache replacement algorithms specifically designed for shared caches [3] may provide relief.

In the future we hope to explore techniques to better identify and respond to the phenomena. Experiments suggest that we may be able to moderate the behavior in software by periodically suspending randomly selected threads for a very brief period. This serves to disrupt and perturb the steady-state and provides statistical performance isolation over the long term by injection of randomized noise.

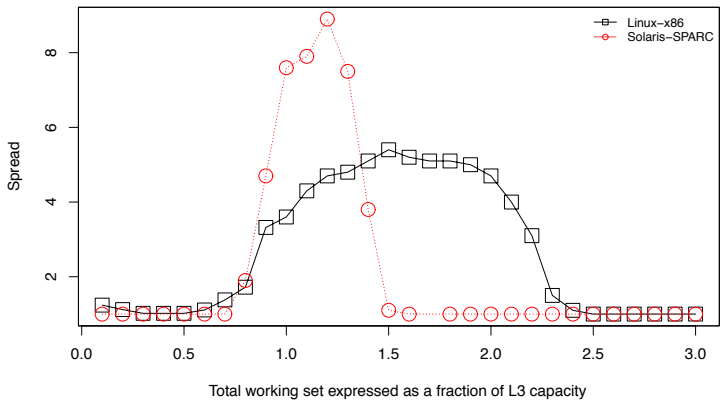

Figure 1: Fairness

Nir Shavit was supported in part by NSF grants CCF-1217921 and CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

4. REFERENCES

[1] Y. Afek, D. Dice, and A. Morrison. Cache Index-aware Memory Allocation. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 55–64, New York, NY, USA, 2011. ACM.

[2] B. Brett, P. Kumar, M. Kim, and H. Kim. CHiP: A Profiler to Measure the Effect of Cache Contention on Scalability. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1565–1574, Washington, DC, USA, 2013. IEEE Computer Society.

[3] A. K. Katti and V. Ramachandran. Competitive Cache Replacement Strategies for Shared Cache Environments. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 215–226, Washington, DC, USA, 2012. IEEE Computer Society.

[4] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[6] Oracle Corporation. Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture, 2012.