Shared Last-Level Caches and The Case for Longer Timeslices

Viacheslav V. Fedorov
Department of Electrical and Computer Engineering
Texas A&M University
College Station, TX, USA
nihrom@tamu.edu

A. L. Narasimha Reddy
Department of Electrical and Computer Engineering
Texas A&M University
College Station, TX, USA
reddy@ece.tamu.edu

Paul V. Gratz
Department of Electrical and Computer Engineering
Texas A&M University
College Station, TX, USA
pgratz@gratz1.com

ABSTRACT

Memory performance is important in modern systems. Contention at various levels in memory hierarchy can lead to significant application performance degradation due to interference. Further, modern, large, last-level caches (LLC) have fill times greater than the OS scheduling window. When several threads are running concurrently and time-sharing the CPU cores, they may never be able to load their working sets into the cache before being rescheduled, thus permanently stuck in the "cold-start" regime. We show that by increasing the system scheduling timeslice length it is possible to amortize the cache cold-start penalty due to the multitasking and improve application performance by 10–15%.

CCS Concepts

•Software and its engineering → Process management; Scheduling; Multiprocessing / multiprogramming / multitasking;

Keywords

Last-level cache; Operating systems; Scheduling; Multiprocessing

1. INTRODUCTION

Last-level cache (LLC) sizes have grown substantially with the continued march of Moore's Law, with LLC sizes up to 45 MB for recent Intel [1] and 32 MB for IBM [2] processors. These large LLC sizes, however, are beginning to challenge commonly held assumptions in operating system (OS) timeslice scheduling behavior. In particular, fill times for large LLCs have grown to the point that they are a significant portion of the application's scheduled timeslice, resulting in lost performance and efficiency, particularly in highly loaded, multi-process, multi-core environments. This problem promises to be exacerbated as new technologies, such as 3D stacking, make extremely large LLCs feasible [3]. In this paper we examine how OS scheduling can be adapted for the large LLC sizes found in today's processors, to reduce contention between applications and improve performance in multi-process, and multi-core environments.

In typical computer systems, the OS coordinates application access to system resources, from I/O devices to time on the processor cores themselves. Current OSes have developed fair scheduling algorithms, such as the "completely fair scheduler" (CFS) in the Linux OS, which ensures all running applications' timeslices together constitute an equal share of running time on the processor(s). To provide the illusion that each application has sole ownership of the processor, timeslices are kept fairly short, with 1–5ms being a typical timeslice interval in current systems. Although, in current multi-processor (multi-core) systems, more than one application or thread can actually run simultaneously on different cores, in many situations, particularly server and cloud implementations, many more processes must execute than there are physical processors available so timeslice scheduling persists.

The concept of timeslices emerged long before the era of modern huge LLCs. Historically, it was safe to assume the cache-fill time for a thread would be insignificant because it constituted only a small fraction of the whole timeslice. While this was a reasonable assumption with LLC sizes $\leq$256KB, with current LLC sizes in the Megabytes, this assumption ceases to be valid. Consider a case where each main memory access takes 200 CPU cycles. Assuming random accesses to memory (not streaming), filling a 4MB LLC with 64-byte lines would require fetching 65,536 lines. This would take about 13M cycles, or 6.5ms for a 2GHz CPU. Comparing this time to a typical OS scheduling timeslice length of 1–5ms, it is clear that modern short timeslices might lead to the performance degradation through LLC interference - the threads just do not have enough time to load their working set into the cache. Moreover, the modern workloads exhibit growing working sets which demand more cache space, further amplifying the effect. The larger numbers of threads found in multicore systems exacerbate

MEMSYS '15 Washington, DC, USA

DOI: 00.000/123_4

the problem since a thread may not be scheduled until all other threads have been scheduled in round-robin fashion, thus increasing the chances that the entire working set of a thread is knocked out of the LLC, even at larger size LLCs. Virtualization contributes to this trend of higher number of simultaneous threads sharing the LLC.

In summary:

- Large modern LLCs require significant time to fill the cache;
- Large numbers of threads in a multicore system lead to cache contention among parallel threads as well as across threads time-sharing the cores;
- Large working sets in modern workloads require more cache space thus take longer to fill the LLC;
- OS scheduling timeslices have traditionally been kept short to provide system interactivity.

These factors unleash severe performance degradation in system workloads. As a result, in real world situations where more threads are being scheduled than there are cores available, each distinct thread (a) experiences a significant cache cold-start effect *every time it is scheduled* on a CPU; (b) is further slowed down by more aggressive, cache-thrashing threads running in parallel; (c) suffers low IPC due to LLC cache misses during the *majority of its timeslice*.

Others have studied the mitigation of this type of contention and interference in shared LLCs. These techniques predominantly rely upon hardware modifications, such as modified cache replacement policies [4, 5, 6, 7] and cache partitioning schemes [8, 9, 10, 11, 12]. Software techniques mainly deal with thread placement [13]. We note that with conventional (16 to 32-way set-associative) cache designs, it is challenging to ensure fair partitioning in many-core systems. Even in an 8-core system with two threads per core, a total of 16 threads share the cache. This leaves about 2–4 ways per thread on an average, and requires costly hardware and repartitioning of the cache every several milliseconds due to the OS context-switching in new threads which may have different cache demands.

Further, much of the prior work does not even *consider* the effect of the OS time-sharing the cores, and instead assumes that each thread has a permanently dedicated core with ample time to reach an LLC steady state (often 1–10 billion instructions simulated with no time-sharing). In the real time-sharing systems, threads run for 1–5ms (∼10 million instructions executed per timeslice), or up to 100× less, before another thread is scheduled on that core. While this behavior is of little consequence for instruction and data access streams of individual threads, the LLC utilization and hit-miss patterns are greatly affected by the fact that several threads contend in the cache in time per core *as well as* across cores.

Unlike prior work which focuses on hardware techniques to deal with LLC contention, in this paper we explore a different approach. We aim to minimize LLC contention via a purely systems software approach. We look at OS scheduling as a possible approach to improving the system performance.

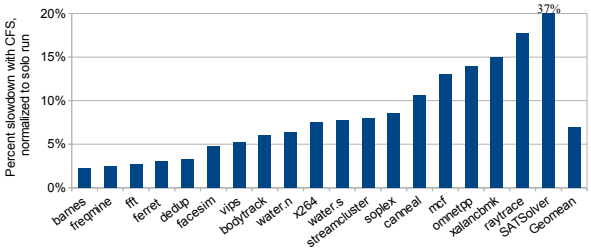


Figure 1: Worst-case slowdown of benchmarks running concurrently with cache aggressive microkernels, with conventional scheduler, normalized to solo run.

2. MOTIVATION

Cache sharing leads to interference in the LLC. To show the worst-case performance degradation due to this sort of interference, Figure 1 presents the throughput degradation of a set of benchmarks taken from the SPEC, PARSEC, and SPLASH benchmark suites, running under the conventional CFS scheduler concurrently with a cache interference inducing microbenchmark each, as compared to running solo. The microbenchmark has a significantly larger working set than the LLC size and it has a high enough access rate that a large fraction of the LLC will be flushed during its timeslice. Under these conditions, we see that the applications experience up to 37% slowdown, and 8% on an average, due to the LLC interference.

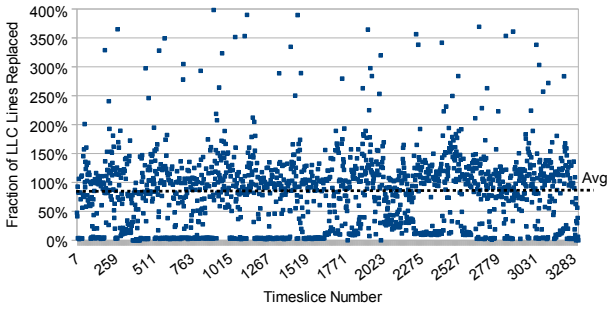
With modern multi-core systems, not only is the cache shared across the threads running in parallel, but it is also time-shared by the threads sharing each distinct core. As a result, the interference is amplified. In an 8-core system with only 4 threads per core, each distinct thread will experience the accumulated effects of interference from 31 other threads when it is scheduled on the core again.

Just how severe is this penalty? If we can estimate what percentage of the LLC gets overwritten during the time a thread is off the CPU, it is possible to tell approximately how much of the thread's working set is retained in the cache between context switches, and thus predict its performance.

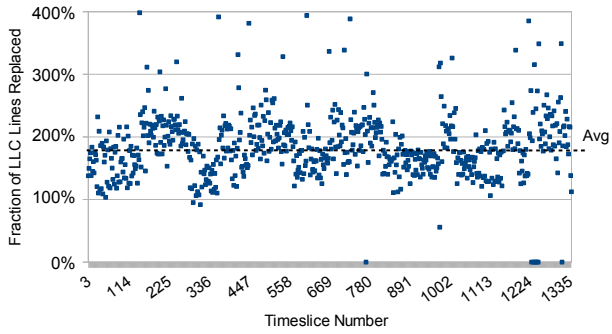
Figure 2 presents the cumulative LLC misses, as a fraction of the total LLC size, caused by interfering threads during the interval while an observed thread is off the CPU. E.g. in a system with four threads per core, this interval would amount to three timeslices. The X axis depicts the samples (every interval between reschedulings), and the Y axis shows the percentage of the cache lines overwritten since the thread's previous timeslice. This data was obtained using the CPU performance counters, for a *SATSolver* application between consecutive context switches [1]. The experiment was performed on a 2-core system with a 4MB LLC (65K cache lines). Results are presented for both two threads per core and four per core (4 threads and 8 threads total, respectively).

The figures indicate that with as little as two applications per core, the whole LLC is effectively fully replaced, with 84% of the cache lines being overwritten on an average be-

[1]Here we measure the amount of cache thrashing due to all the other threads in the system, thus application choice is irrelevant.



(a) Dual-core system with 2 applications/core



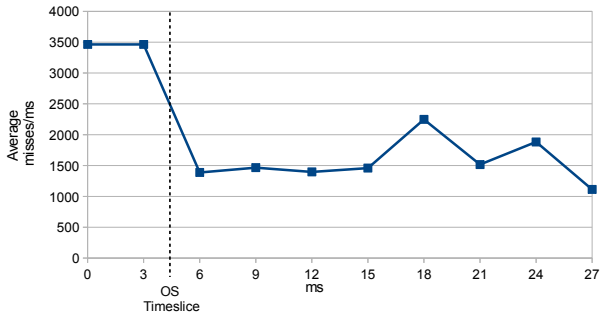
(b) Dual-core system with 4 applications/core

Figure 2: Number of LLC Misses between reschedulings, as a percentage of LLC lines replaced.

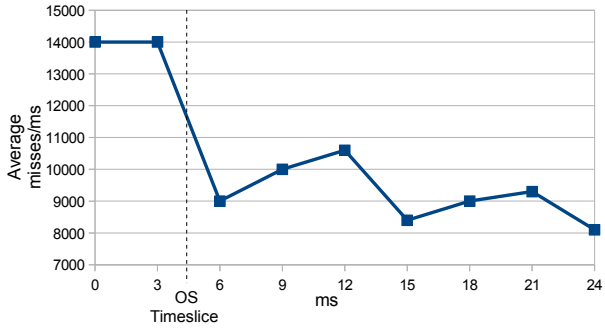
tween thread's rescheduling. Worse, with four applications per core (Figure 2b), the LLC is fully replaced in all of the cases. Even taking into account the possibility of cache interference (when several threads have misses on the same cache line), the observation that 170% of the LLC lines are overwritten, enables us to expect with high confidence that once a thread is context-switched off the CPU, it then returns to a cold cache.

Further analysis, using data access traces obtained with the PIN toolkit [14], shows that, between two consecutive schedulings, each thread re-accesses an average of about 75% of the data it touched during the preceding timeslice. These results indicate that (a) **every** thread starts with a cold cache after **every** context switch, and (b) by increasing timeslice length, we could allow threads enough time to more effectively utilize the cached portion of their working set, thus amortizing the cold cache effect. An alternative approach of partitioning the cache to keep the working sets from getting thrashed, requires extra hardware and, as we show later, does not provide a substantial performance gain beyond what timeslice aggregation can achieve.

The LLC interference is aggravated by short OS scheduling timeslices. Once context-switched out, and then rescheduled back onto a core, a thread does not execute long enough to fully load its working set into the cache and is thus forced to run with high cache miss rate (low IPC). Figure 3 illustrates this point. Figure 3a shows the LLC Misses per *ms* for a thread running the *Raytrace* application in a time-shared multicore system inside a single, extended time slice. Figure 3b shows analogous data for a thread running *Xalancbmk* application. The data was sampled every 3ms using the CPU performance counters. We note that, during the first 3ms of execution (which is approximately the normal OS timeslice length in this case), each thread experiences a high number of misses due to the cold cache effect described. Note, after the working set is filled into the LLC (from *6ms* on), the thread experiences a much lower and stable miss rate. Thus at least $3-5$ *ms* of execution is required before cold misses become insignificant for **every timeslice**. At least 2–3× that time is needed to amortize the high initial miss rate. Unfortunately, in a conventional system, the threads would be de-scheduled from the core after ∼5ms of execution. Utilizing longer OS timeslices, however, the system can amortize the cold cache effect of context switches, allowing it to more effectively leverage the LLC space, boosting the thread performance and decreasing the memory bus contention due LLC misses.



(a) Raytrace



(b) Xalancbmk

Figure 3: Misses/ms for an extended timeslice, sampled every 3ms

One potential side-effect of increased time slices is that threads will more thoroughly fill the LLC, which in turn, increases the temporal cache thrashing with respect to all the other threads which run later. However, because the cache data retention for each distinct thread between schedulings is very low to begin with (i.e. <10%), this side-effect is far outweighed by the improvement in LLC miss rate.

3. DESIGN

Our design is illustrated on the Figure 4. The figure shows a snapshot of the three types of OS schedules on one core, with two threads being scheduled. We start from a conventional CFS timeslice schedule, as shown in the top of the figure. Note how after each context switch, threads are

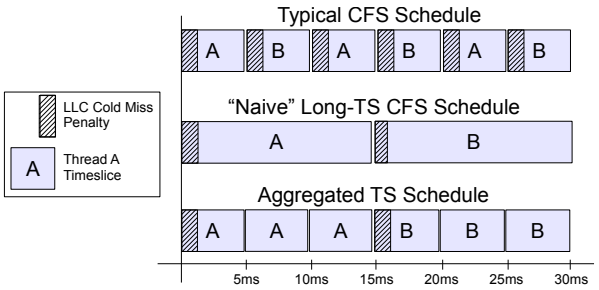


Figure 4: A comparison of the conventional CFS timeslices vs proposed aggregated timeslices.

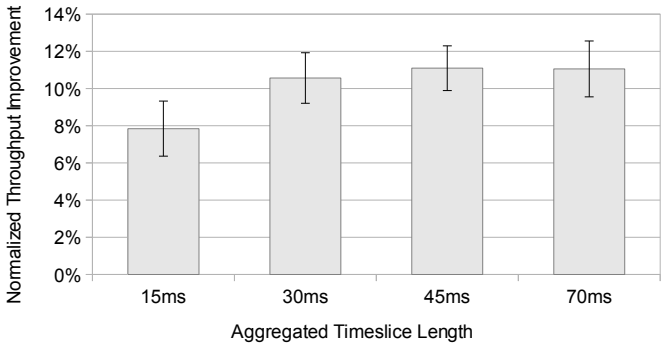


Figure 5: Runtime improvement versus CFS baseline in a dual-core system with 16MB LLC. Geomean across sixteen application mixes, four applications per core.

paying the cold cache penalty, hence suffering degraded performance.

The naïve approach to increasing the OS timeslice length (illustrated in the middle of Figure 4) is to tweak the constants in the Linux kernel. Note that the cold cache penalty is amortized across the longer thread running times. This approach is, however, imprecise and requires frequent retuning in order to maintain the desired timeslice length. The scheduler constants only provide an upper bound for the scheduling slices, and when the system load changes, the kernel automatically shortens the timeslices in response to the increasing number of threads in the system.

An alternative approach, presented in the bottom of Figure 4, which we call *timeslice aggregation*, is to modify the scheduler code in such a way that it would consecutively reschedule a given thread for a set amount of time before switching to the next thread (selected utilizing the long-run time sharing fairness of the Linux CFS scheduler). In such a scheme, instead of having one long, monolithic timeslice, several timeslices are effectively aggregated. This allows precise timeslice control, as well as allowing the scheduler to switch to a higher-priority thread (such as an interrupt bottom-half), if needed, without having to wait for the current low-priority thread to finish its full extended timeslice. Aggregation guarantees low impact on system interactivity, although it may provide lower performance boost in highly-interactive systems with frequent interrupts.

4. EVALUATION

4.1 Methodology

We implemented our proposed the timeslice aggregation scheme in the Linux kernel. Standard Linux timeslice length was used as a baseline. Note, in Linux, timeslice length varies between 1ms and 15ms dependent on system load (number of threads running) and number of CPUs in the system, with shorter time slices as the load increases.

We ran our experiments on an an Intel Xeon E5 with 16MB LLC and 90GB of RAM. We emulated double- and quad-core configurations by pinning the threads to specific cores. Additionally, we ran experiments on a smaller system, an Intel Core 2 duo E6550 processor with 4MB last-level cache, and 8GB of RAM, in order to gain some insight on how the LLC size affects the performance degradation due to thread contention. We utilized memory-oriented applications from the PARSEC 2.1, SPLASH-2x, and SPEC CPU 2006 suites [15, 16, 17] running with their native (or equivalent) input sets. Namely, barnes, bodytrack, canneal, dedup, facesim, ferret, fft, freqmine, mcf, omnetpp, raytrace, soplex, streamcluster, vips, water.nsquared, water.spatial, x264, and xalancbmk. Additionally, we used a SATSolver benchmark with an input set from a SAT contest [18]. For these experiments, we selected those applications which have working sets larger than the L2 cache, *i.e.* those which show at least 3% performance degradation when co-scheduled with a memory-intensive microbenchmark. For two-core experiments, we arranged 16 mixes of four threads per core, with a total of eight threads per mix. In four-core experiments, we utilized four threads per core for a total of sixteen threads per mix.

4.2 Performance Improvement

Threads running in a multi-core multi-process system experience severe LLC interference. One of the major implications of this interference is that each thread starts each timeslice with a cold cache. By allowing threads to run longer before a context switch, we can effectively amortize the performance lost due to cold cache startup, thus boosting the system throughput without any hardware modifications.

We present two sets of experiments on a quad-core machine with 16MB LLC. First set consists of 16 mixes of 4 applications per core, run on two cores, for a total of 8 applications per mix (applications are pinned so two cores are held idle). The second set consists of 6 mixes of 4 apps per core, run on four cores, for a total of 16 applications per mix. In both experiments, we compare the geometric mean runtime improvement of the applications with four aggregated timeslices (15ms, 30ms, 45ms, and 70ms) versus the conventional CFS scheduler short timeslice (∼4ms).

Figure 5 shows the geometric mean runtime improvement figures for application mixes in two-core experiments, normalized to a system with conventional timeslices. The error bars depict the variance in performance across the applications (±1 standard deviation). We observe a gain of 8–11% on average. With the large 16MB LLC, it takes a considerable amount of time for applications to load their work-

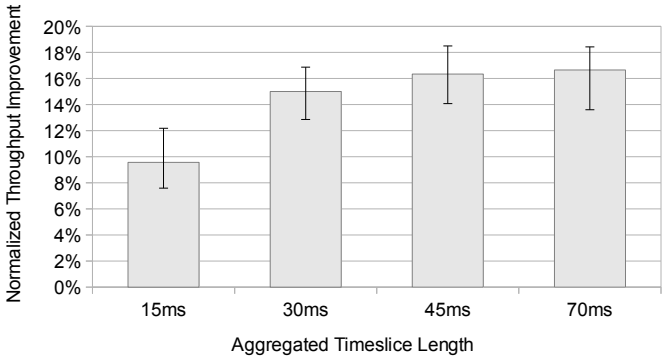


Figure 6: Runtime improvement versus CFS baseline in a quad-core system with 16MB LLC. Geomean over six application mixes, four per core.

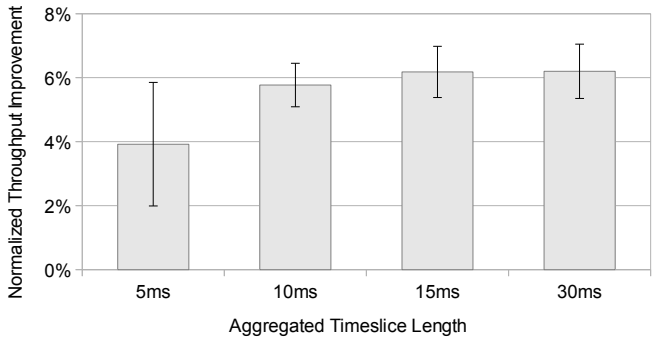


Figure 7: Runtime improvement over CFS baseline in a dual-core system with 4MB LLC. Geomean across sixteen application mixes, four applications per core.

ing sets in the cache, thus the performance improvement increases from 15ms to 30ms timeslice length due to the increased amount of time to amortize LLC cold start cache misses. However, beyond 30–45ms timeslices, there is only marginal performance improvement since the applications' working set data is then loaded in the LLC and the cold start penalty amortized.

Figure 6 presents the runtime improvement figures for application mixes in a quad-core system, normalized to a system with conventional scheduler. The error bars depict ±1 standard deviation. Again, we observe diminishing returns on throughput improvement beyond 45ms timeslices. Here the benefit is slightly larger than in the dual-core case, in part due to the more thorough replacement caused by running more simultaneous applications.

4.3 Effect of LLC Size on Optimal Timeslice Length

Intuitively, larger caches take longer to fill than smaller ones, so the threads need longer timeslices when running with a larger LLC, in order to amortize the cold cache penalty. We conducted the same set of two-core experiments as in Section 4.2 in a true two-core system with 4MB cache. Figure 7 presents the geometric mean runtime improvement figures, for sixteen application mixes in a dual-core system with aggregated timeslice lengths from 5 to 30ms. Each bar reflects the performance improvement normalized to running the mixes on a system utilizing the conventional CFS Linux scheduler with standard timeslices (∼2ms in this case due to a smaller number of cores in the system and initial CFS settings). Again, the error bars depict ±1 standard deviation.

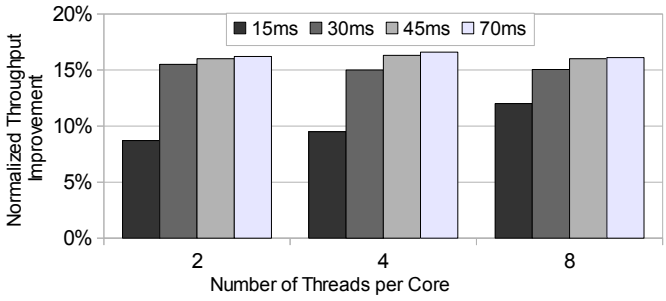


Figure 8: Runtime improvement over CFS baseline with 2, 4, and 8 applications per core, in a quad-core system.

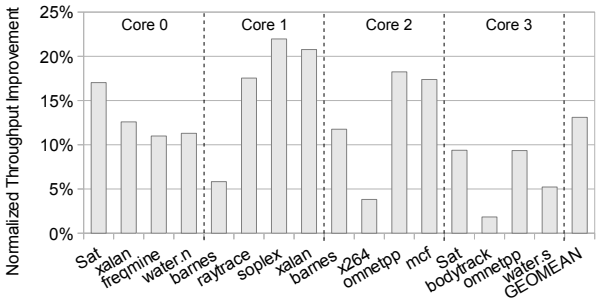


Figure 9: Detailed runtime improvement, 45ms aggregated timeslices versus CFS baseline in a quad-core system with 16MB LLC. Four applications per core, sixteen total.

In the figure we see that here timeslice aggregation with 5ms-long timeslices yields about 4% geomean runtime improvement, while 15ms-long timeslice aggregation boosts the performance by 6.2%. Versus the previous results with a large LLC (Section 4.2), we note that: (a) the overall performance improvement is less, and (b) the performance benefit saturates at a shorter relative timeslice length. The differences primarily derive from the fact that a smaller cache takes less time to refill on cold restart than a larger cache, thus the total performance penalty is less and it takes less time to amortize (about 15ms versus 30-45ms with larger LLC). We conclude that the optimal timeslice length for a given system depends on the size of the LLC.

With caches getting larger and the number of threads increasing, the data seems to suggest that longer timeslices are needed to amortize the cold start penalty and improve the system performance.

4.4 Performance vs Application Count

Here we examine sensitivity to the number of applications per core. Figure 8 shows the results for experiments with two, four, and eight applications per core. Across all experiments, the performance improvement saturates at ∼15% for aggregated timeslices larger than 30ms. This is in line with

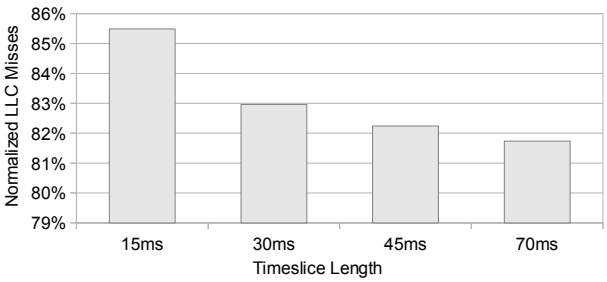


Figure 10: LLC misses with aggregated timeslices, normalized to CFS in a quad-core system with 16MB LLC.

our reasoning that shorter timeslices are not quite sufficient to amortize the cold cache penalty. The slightly better results in 8 thread-per-core experiments can be explained by the fact that additional threads more thoroughly replace the contents of the LLC, leading to more performance loss on cold restart.

5. DISCUSSION

Figure 9 presents the per-application runtime improvement for one application mix in a quad-core system, with 45ms aggregated timeslices. The figure shows that all the applications benefit from longer timeslices, with an average improvement for this particular of ∼12%.

5.1 LLC Miss Reduction

Figure 10 presents the total number of LLC misses for one mix of applications with aggregated timeslices, as a fraction of misses with conventional CFS. We observe that timeslice aggregation helps alleviate about 15% to 18% of the total LLC misses. Note that these misses are forced cold cache misses due to multitasking, hence reducing them leads to pronounced improvement in system throughput, as we have seen in Section 4.2. The geometric mean LLC miss reduction per application is 30% (we do not present the figures for brevity). Again, we observe that about 30-45ms timeslice length is enough to amortize the cold start penalty, and beyond that we hit diminishing returns (*i.e.* the rest of the misses are conflict and compulsory misses intrinsic to the application mix running in the system).

5.2 Cache Partitioning vs Timeslice Aggregation

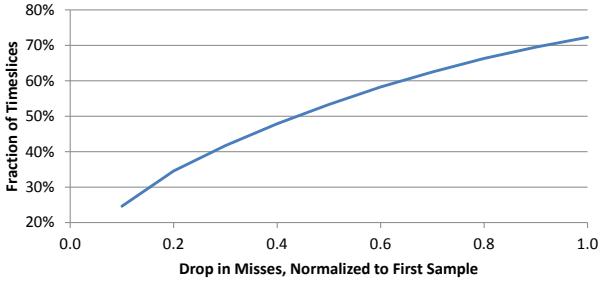
We notice that timeslice aggregation scheme improves the application performance by 10–15% in multi-core systems. The interesting question is, can this technique be augmented by partitioning the cache? Longer timeslices help to amortize the cold cache effect, but they do not prevent it. With cache partitioning, each thread would retain its cache contents across timeslices thus avoiding the cold-start effect completely. The catch here is that it is challenging to effectively partition a 16-way cache among 10–20 threads. Further, a partitioned cache gives a fraction of its space to each application, while with timeslice aggregation, all the cache is available to the application once it has been refilled. Intuitively, splitting applications into groups based on their cache size demands, one would need to enforce partitioning for a much smaller number of threads at a time (*i.e.* partition the cache into 4 pieces where 3 applications get 1 full piece each, while the rest of the applications share a single partition; rotate the partition assignment every 200ms).

Here we analyze an ideal case, assuming that the cache is equipped with a partitioning scheme which is able to split the cache among all the threads in the system, based on cache utility or working set sizes, etc., such that each thread achieves the maximum benefit. We gathered the statistics of the Miss rate fall-off behavior among four mixes of applications. Looking at Figure 3a as an example, let us take the first data point (at 3ms) as the maximum (100%). Now, the fifth data point (at 15ms) in this example is $\alpha = 0.42$ of the maximum miss rate. We are interested in the fraction of misses that could be avoided if a perfect partitioning scheme was implemented in the cache. Let us assume a best-case scenario where the number of misses is high during the first 3ms, but rapidly drops and stays at the level of the 15ms point. We can approximate the total number of misses the thread encountered in 15ms as follows: $3ms \times Xmisses + 12ms \times \alpha Xmisses = (3ms + 12ms \times \alpha) \times X$. Now with perfect partitioning, there would be no spike in misses during the first 3ms, thus in this case the number of misses in 15ms is: $15ms \times \alpha Xmiss/ms$.

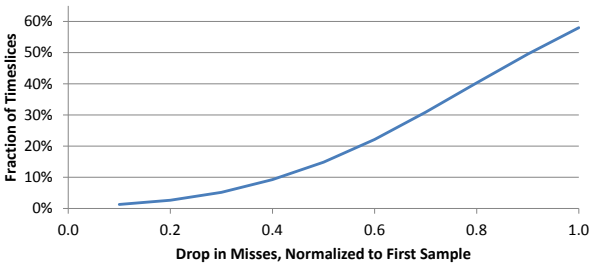
The improvement in cache misses with cache partitioning beyond 15ms aggregated timeslices can be thus calculated as follows: $1 - \frac{15\alpha}{3+12\alpha} = 1 - \frac{5\alpha}{1+4\alpha} = \frac{1-\alpha}{1+4\alpha}$. Note that the multiplier 4 in the denominator increases with the increase of aggregated timeslice length, thus the maximum miss rate improvement with long timeslices is decreased.

Figure 11 depicts the CDF functions for the two most representative applications from our application mixes. The two charts on the top show, for each α, the fraction of timeslices that the thread experienced a corresponding drop in misses from the 3ms to the 15ms data point. For instance, with *Freqmine* benchmark (Figure 11a), the miss rate drop is smaller than $\alpha = 0.6$ for 40% of the timeslices, and smaller than $\alpha = 0.9$ for 30% of the timeslices. The charts on the bottom reflect the corresponding estimated improvement in total number of misses with the data obtained from the top charts. Continuing the *Freqmine* example: with partitioning, we would alleviate 90% misses (α=0.1 drop in Figure 11a), which is $\frac{1-\alpha}{1+4\alpha} = 0.64\times$ better than 15ms aggregated scheme, for 25% of timeslices. The timeslices with α=0.1 drop with partitioning would contribute to 16% misses alleviation beyond the 15ms scheme (Figure 11c). Then, 80% misses (0.44× better than 15ms scheme) for additional 10% of timeslices, which would contribute to additional 6% misses alleviation beyond the 15ms scheme. In other words, we integrate the upper graph with the $\frac{1-\alpha}{1+4\alpha}$ multiplier, obtaining a total of about 27% Misses reduction beyond the 15ms scheme. Similarly, *Soplex* application would achieve 7% Misses reduction with cache partitioning, beyond the 15ms scheme. The more timeslices exhibit the larger drop in miss rate, the more can be gained with partitioning.

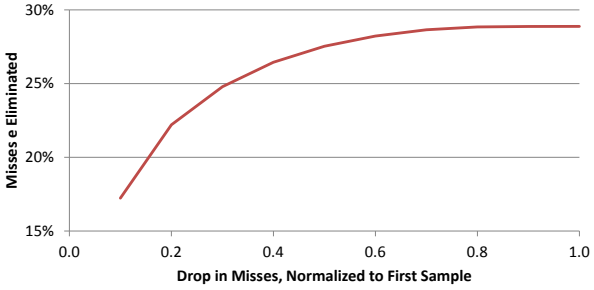
Our estimates show that for most of the applications, partitioning provides only about 5–10% additional benefit beyond 15ms aggregated timeslices. With longer timeslices,



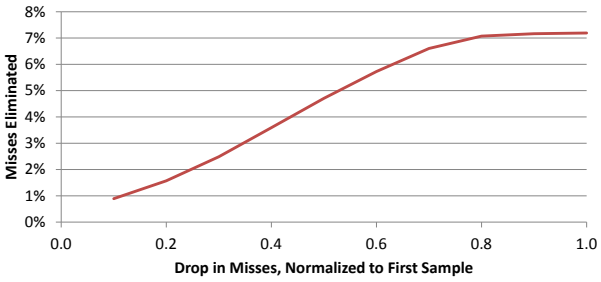
(a) Freqmine cold cache effect.



(b) Soplex cold cache effect



(c) Freqmine miss improvement beyond 15ms aggregation scheme



(d) Soplex miss improvement beyond 15ms aggregation scheme

Figure 11: CDF functions for cold cache effect, and estimated LLC miss improvement beyond 15ms aggregation scheme.

the added gains rapidly drop. Our results thus indicate that partitioning does not seem to offer substantial additional benefit beyond longer time slices. We further note that time slice aggregation is a purely software solution without any additional hardware support, thus the overhead of implementing this scheme is much lower than partitioning. With partitioning, however, there is no cross-core interference, *i.e.* the steady state miss rates in Figures 2a and 2b would be lower with cache partitioning. Generally we find these applications are not particularly sensitive to inter-core interference, with only a ∼5% miss rate increase when run simultaneously with other applications.

5.3 Fairness

Since our scheme is based on the Completely Fair Scheduler, by increasing the timeslice length, the system fairness should not be affected. We observe that by aggregating the timeslices, the system achieves similar fairness among the applications. Note however that the aggregated timeslices allow fairness over longer time scales than the original CFS, while potentially being unfair over timescales smaller than the aggregated scheduling times.

5.4 Dynamic Timeslice Extension

A natural extension of our approach is to dynamically control the timeslice length, adapting to the application execution phases. The intuition is that overlong timeslices can cause more LLC interference where applications on different cores will fight for the cache space. We implemented a dynamic scheme as a modification of the CFS scheduler in Linux Kernel.

The basic idea of our dynamic scheduler was to monitor the threads' LLC fill rate and individually adjust the timeslice aggregation so that each thread is only able to fill a preset fraction of the cache. We leveraged the kernel interface to the CPU performance counters to compute the MPTS (misses per timeslice) metric. The overhead of accessing the counters is approximately 2 μsec per read. Compared to the time one scheduler invocation and context switch takes, this overhead is negligible.

Because the dynamic scheme is an extension of CFS, regardless of each thread's timeslice length, all threads are given a fair share of CPU time. All runnable threads are stored in a tree where the key is their virtual runtime length, *vruntime*. After a thread has been scheduled and finished its timeslice, the *vruntime* variable is incremented by the timeslice length, and the thread gets put back into a tree. A new thread with the lowest *vruntime* is picked from the tree. When we are aggregating the timeslices for a thread, after one scheduling of such thread, its *vruntime* variable will get much larger than other threads' *vruntimes*. As a result the thread will remain unscheduled for a longer time. Thus the scheduler automatically takes care of fairness.

Our experiments show that, while the dynamic scheme is adapting to individual applications' behavior, it does not provide any noticeable performance improvement beyond the Systemwide aggregation. We calculated the average timeslice lengths produced by the dynamic scheme, and based on that data, it is marginally better than Systemwide (by about 0.7–1% on an average) when the timeslices are shorter than 6 ms, but beyond that, there appears to be no gain from adaptivity.

5.5 Even Larger LLCs

It is tempting to think that with larger LLCs multiple working sets might be able to fit simultaneously and hence

the benefits from longer timeslices may not persist. First, our results show that longer timeslices benefit both 4MB and 16MB LLCs, with the trend towards more benefit in larger LLCs. Second, application sizes and data sets tend to grow as larger systems become available. Virtualization and server consolidation continues the trend of increasing resource sharing and the resultant contention at the LLC. As we have seen (Figures 2a and 2b), four applications in a system are able to overwrite a 4MB LLC leaving each thread with a cold cache every timeslice. Similarly, in an 8-core system, with four threads per core, a 32MB cache may easily be overwritten. Note that, with larger cache sizes come longer refill latencies, aggravating the cold start penalty. We thus expect the timeslice length required to efficiently amortize the cold cache penalty in systems with large LLCs to be even greater.

6. CONCLUSION

System performance in time-shared, multitasking systems is degraded by the cold restart fill time after every context switch seen in modern, large, last-level caches. Modern cache sizes have grown so large, and the OS scheduling timeslices so small, that the cache fill time is greater than the scheduling window. It is possible to amortize the cold cache penalty by allowing the applications to run longer between reschedulings. Here we present a simple scheme for timeslice aggregation, and show that it is possible mitigate the system slowdown due to the multitasking, boosting the throughput while observing fairness.

Acknowledgement

This research is supported in part by the National Science Foundation, under grants #1320074 and #1439722.

References

[1] "Intel haswell." http://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz. 2014.

[2] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "Ibm power7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1:1–1:29, 2011.

[3] G. H. Loh, "Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 201–212, IEEE, 2009.

[4] N. P. Jouppi, "Cache Write Policies and Performance," in *ISCA*, 1993.

[5] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 175–186, IEEE Computer Society, 2010.

[6] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," in *SIGMETRICS*, 1999.

[7] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive Caches: Effective Shaping of Cache Behavior to Workloads," in *MICRO*, 2006.

[8] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[9] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *In Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 544–555, 2004.

[10] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 57–68, ACM, 2011.

[11] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, pp. 7–26, Apr. 2004.

[12] D. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Trans. Comput. Syst.*, vol. 5, pp. 305–329, Oct. 1987.

[13] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, pp. 4:1–4:28, Dec. 2012.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[16] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 47–56, Sept 2008.

[17] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[18] G. Audemard and L. Simon, "Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (system description)," in *Pragmatics of SAT 2012 (POS'12)*, jun 2012. dans le cadre de SAT'2012.