On the Energy Efficiency of Last-Level Cache Partitioning

Abstract

Computing systems frequently have a mix of interactive, real-time applications and background computation to execute. In order to guarantee responsiveness, the interactive and background applications are often run on completely disjoint sets of resources to ensure performance isolation. These practices are expensive in terms of battery life, power and capital expenditures. In this paper, we evaluate the potential of hardware cache partitioning mechanisms and policies to provide energy efficient operating environments by running foreground and background tasks simultaneously while mitigating performance degradation. We evaluate these tradeoffs using real multicore hardware that supports cache partitioning and energy measurement. We find that for modern, multi-threaded benchmarks there are only a limited number of application pairings where cache partitioning is more effective than naive cache sharing at reducing energy in a race-to-halt scenario. However, in contexts where a constant stream of background work is available, a dynamically adaptive cache partitioning policy is effective at increasing background application throughput while preserving foreground application performance.

1. Introduction

Energy efficiency and predictable response times are first-order concerns across the entire computing spectrum, ranging from mobile clients to warehouse-scale cloud computers.

For mobile devices, energy efficiency is critical, as it affects both battery life and skin temperature, and predictable response times are essential for providing a fluid user interface. Some mobile systems have gone so far as to limit which applications can run in the background [1] to preserve responsiveness and battery life, despite the obvious concerns this raises for user experience.

In warehouse-scale computing, energy inefficiencies are felt in the operational costs of consuming electricity and can significantly impact the capital costs of the infrastructure needed to distribute power and cool the servers [3, 17]. Researchers have also found unpredictable response times to be very expensive in warehouse-scale computing [30]. In one example, inserting delays in a search engine slowed down user response time by more than the delay, which decreased the number of overall searches, user satisfaction, and revenue; the results were so negative that the researchers stopped their experiment early [30]. To preserve responsiveness, cloud computing providers often dedicate large clusters to single applications, despite the hardware being routinely utilized at only 10% to 50% [3].

Due to the diminishing returns from hardware techniques to extract further instruction-level parallelism from single-threaded code and power concerns caused by the end of traditional transistor scaling, all platforms have now moved to multicore processors [2]. Application workloads already include parallelized codes, and we expect this fraction to increase over time. However, few applications scale perfectly with increasing core count, leading to underutilized resources.

In both client and cloud scenarios, it is desirable to make use of the underutilized resources to schedule background tasks to improve overall throughput and energy efficiency, but only if this results in minimal disruption of latency-sensitive user-facing tasks. In the client, the goal is to complete background tasks while the foreground task is active, so that the mobile device can more quickly go into a very low-power hibernation mode and thereby extend battery life. In the cloud, the goal is to obtain the greatest value from the huge sunk investment in machines, power distribution, and cooling.

The obvious danger in co-scheduling computational tasks to achieve better utilization is that low-priority background tasks can degrade the responsiveness of high-priority foreground ones by impacting any shared hardware resources, such as on-chip shared cache or off-chip DRAM bandwidth. Mechanisms to mitigate such degradation are the subject of active research [16, 23]. In particular, techniques for *partitioning* the capacity of a shared *last-level cache* (LLC) have received much attention [16, 20, 28, 32]. However, past work has mostly been simulation-based, which limits the size of applications considered, and has predominantly considered multiprogramming of a limited set of sequential applications.

In this paper, we use a real commercial multicore processor that includes an experimental hardware LLC partitioning mechanism to explore the potential benefits of last-level cache partitioning using multiple large parallel applications taken from several modern benchmark suites.

We first characterize three modern benchmark suites in terms of their scalability, cache capacity requirements, and sensitivity to interference with respect to shared hardware resources. We measure not only the performance impact of different resource allocations on the benchmarks, but also their effect on socket and overall system energy consumption. We show that many applications have multiple different resource allocations that provide close to the best runtime and lowest energy usage, providing opportunities to transfer resources away from the foreground application to support co-scheduling.

We next take a subset of representative applications and explore how these co-scheduling opportunities play out in practice. We evaluate preserving one application's performance while maximizing the throughput of a background application (as in a cloud), as well as finishing a finite set of tasks before entering a low-power state (as on a client). Our study further characterizes the benefit of introducing increasingly sophisticated policies to manage the LLC partitioning.

Our results show that doing nothing to protect foreground applications while co-scheduling tasks results on average a slowdown of 10%, with the full range being between 1% and 82%. Compared to time-multiplexing tasks across the whole machine, we show it is possible to use co-scheduling to get an average improvement in energy by 12 %, with the full range being between -14% and 37 %. However, while most applications do have the requisite qualities to make co-scheduling effective, only a subset significantly benefit from software-managed partitioning of the LLC. The resource usage of both the foreground and background tasks must be considered when deciding which partitioning strategy to deploy. We also investigate deploying a dynamic repartitioning framework which improves background application throughput by 22% on average, with the full range between 0% and 322%. While the paper does not evaluate server hardware, we believe this approach and resulting conclusions for scheduling philosophy is just as relevant for the cloud, although the quantitative results would surely change.

2. Experimental Methodology

In this section, we describe our hardware platform and the benchmarks we use in our evaluation.

2.1. Platform Configuration

In this paper, we use a prototype version of Intel's Sandy Bridge x86 processor to collect results on resource allocation and application co-scheduling. By using a real hardware prototype, we are able to run full applications for realistic time scales and workload sizes, and while running a standard operating system. The processor is similar to the commercially available client chip, but with additional hardware to support way-based cache partitioning [] in the last-level cache (LLC).

The Sandy Bridge client chip has four quad-issue out-of-order superscalar cores, each of which supports two *Hyper-Threads* using simultaneous multithreading [18]. Each core has private 32 KB instruction and data caches, as well as a 256 KB private non-inclusive L2 cache. The LLC is a 12-way set-associative 6 MB inclusive L3 cache, shared among all cores using a ring-based interconnect. All three cache levels are write-back. Larger server versions of the same processor family have up to 15 MB of LLC capacity.

The cache partitioning mechanism is way-based and modifies the cache-replacement algorithm. Each core can be assigned a subset of the 12 ways in the LLC. Although all cores can hit on data stored in any way, a core can only replace data stored in one of its assigned ways. Allocation of ways among cores can be completely private, completely shared, or overlapping. Data is not flushed when the way allocation changes; newly fetched data will just be written into one of the assigned ways according to the updated allocation configuration.

We use a customized BIOS that enables the cache partitioning mechanism, and run unmodified Linux-2.6.36 for all of our experiments. We use the Linux `taskset` command to pin each application to subsets of the available HyperThreads.

2.2. Performance and Energy Measurement

To measure application performance, we use the `libpfm` library [11,26], built on top of the `perf_events` infrastructure introduced in Linux 2.6.31, to access various performance-monitoring counters available on the machine [19].

To measure on-chip energy, we use the energy counters available on Sandy Bridge to collect the energy used by the entire socket and also the total combined energy of cores, their private caches, and the LLC. We access these counters using the Running Average Power Limit (RAPL) interfaces [19]. The counters measure power at a $1/2^{16}$ second granularity.

In addition, we use a FitPC external multimeter to measure the power consumed at the wall socket by the entire system, at a 1 second granularity. Total system power is generally between 185 W and 240 W. The system-level values are correlated with data collected from the hardware energy counters using time stamps. We observed less than one second of delay in these measurements consistently across all experiments. Together, these mechanisms allow us to collect accurate energy readings over the entire course of an application's execution.

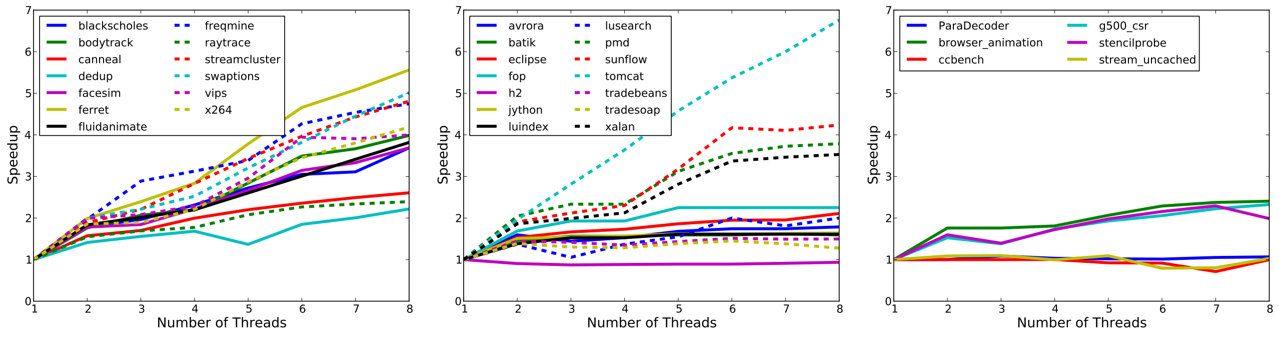
2.3. Description of Workloads

We built our workload using a wide range of codes from three different popular benchmark suites: SPEC CPU 2006 [31], DaCapo [6] and PARSEC [5]. We included some additional application benchmarks to broaden the scope of the study, and some microbenchmarks that exercise certain features of the system.

The **SPEC CPU 2006** benchmark suite [31] is a CPU-intensive, single-threaded benchmark suite, designed to stress a system's processor, memory subsystem and compiler. Using the similarity analysis performed by Phansalkar et al. [27], we subset the suite, selecting 4 integer benchmarks (astar, libquantum, mcf, omnetpp) and 4 floating-point benchmarks (cactusADM, calculix, lbm, povray). Based on the characterization study by Jaleel [21], we also pick 4 extra floating-point benchmarks that stress the LLC: GemsFDTD, leslie3d, soplex and sphinx3. When multiple input sets and sizes are available, we pick the single *ref* input indicated by Phansalkar et al. [27]. SPEC is the only benchmark suite used in many previous characterizations of LLC partitioning [16, 28, 32].

The **DaCapo** benchmark suite is intended as a tool for Java benchmarking, consisting of a set of open-source, real-world applications with non-trivial memory loads, including both client and server-side applications. We used the latest 2009 release. The managed nature of the DaCapo runtime environment has been shown to make a significant difference in some schedulding studies [12], and is also representative of the increasing relevance of such runtimes.

The **PARSEC** benchmark suite is intended to be representative of parallel real-world applications [5]. PARSEC programs use various parallelization approaches, including data- and



(a) PARSEC applications (b) DaCapo applications (c) Parallel apps and microbenchmarks

Figure 1: Normalized speed up as we increase the number of threads allocated to each application.

task-parallelization. We use the version of the benchmarks parallelized with the `pthreads` library, with the exception of `freqmine`, which is only available in OpenMP. We used the full native input sets for all the experiments. Past characterizations of PARSEC have found it to be sensitive to available cache capacity [5], but also resilient to performance degradation in the face of intra-application sharing of caches [38].

We added four **additional parallel applications** to help ensure we covered the space of interest: *Browser_animation* is a multithreaded kernel representing a browser layout animation; *G500_csr* code is a kernel performing breadth-first search of a large graph for the Graph500 contest, based on a new hybrid algorithm [4]; *Paradecoder* is a parallel speech-recognition application that takes audio waveforms of human speech and infers the most likely word sequence intended by the speaker; *Stencilprobe* simulates heat transfer through a fluid using a parallel stencil kernel over a regular grid [22].

We also added two **microbenchmarks** that stress the memory system: *stream_uncached* is a memory and on-chip bandwidth hog that continuously brings data from memory without caching it, while *ccbench* explores arrays of different sizes to determine the structure of the cache hierarchy.

3. Performance Characterization

Our first set of experiments explore the sensitivity of all of the applications to different resources in the system: the number of assigned hyperthreads, the allocated LLC capacity, the various prefetchers, and the on-chip LLC bandwidth and off-chip DRAM bandwidth. We then use machine learning to cluster applications based on their resource requirements, and select a set of representative applications for further evaluation.

3.1. Thread Scalability

We begin with a study of parallel scalability when increasing the number of threads for a fixed problem size. Figure 1 shows the speedup of each application as we increase its allocation from 1 to 8 threads. When adding new threads, we first assign both hyperthreads available in one core before moving on to the next core. For example, the allocation with four threads corresponds to running on both hyperthreads in two cores. This allocation strategy fits our scenario of consolidating applications in a multiprogrammed environment, where different applications should be pinned to different cores to avoid thrashing of the inner levels of the cache hierarchy [35].

Table 1: Summary of thread scalability

Suite	Low scalability	Saturated scalability	High scalability
PARSEC	–	canneal, dedup, raytrace	blackscholes, bodytrack, facesim, ferret,, vips, x264, fluidanimate, freqmine, streamcluster, swaptions
DaCapo	h2, tradebeans, tradesoap	avrora, batik, eclipse, fop, jython, luindex, lusearch	pmd, sunflow, tomcat, xalan
SPEC	all	–	–
μbench-marks	ccbench, paradecoder, stream uncached	browser_animation, g500, stencilprobe	–

Many PARSEC applications scale well (Fig. 1a): six benchmarks scale up over 4×, four benchmarks between 3–4×, and just three show more modest scaling factors (2–3×). For the majority of these applications, we can see that performance does not saturate after a particular number of threads, and keeps growing at a similar rate throughout. This scaling is not the seen for the DaCapo applications in Fig. 1b, which are mostly less scalable than the PARSEC applications. In this suite, only two applications show speedups over 4×, two between 2–3×, and ten between 1–2.3×. Furthermore, the performance of all the applications that do not scale well saturates after 4 or 6 threads. The intrinsic parallelism available in some of the DaCapo benchmarks together with the scalability bottlenecks for garbage collectors explain this behavior [15]. Finally, the scalability results for the additional parallel applications and microbenchmarks are presented in Figure 1c. The microbenchmarks are single-threaded (ccbench and stream_uncached), while the parallel applications are all memory-bandwidth-bound on this platform (we have observed parallel speedups on other platforms), which explains the limited scalability of these benchmarks.

We now classify applications according to their scalability with different number of threads. Table 1 groups applications in each suite into three categories: applications with low scalability, applications that scale up to a reduced number of

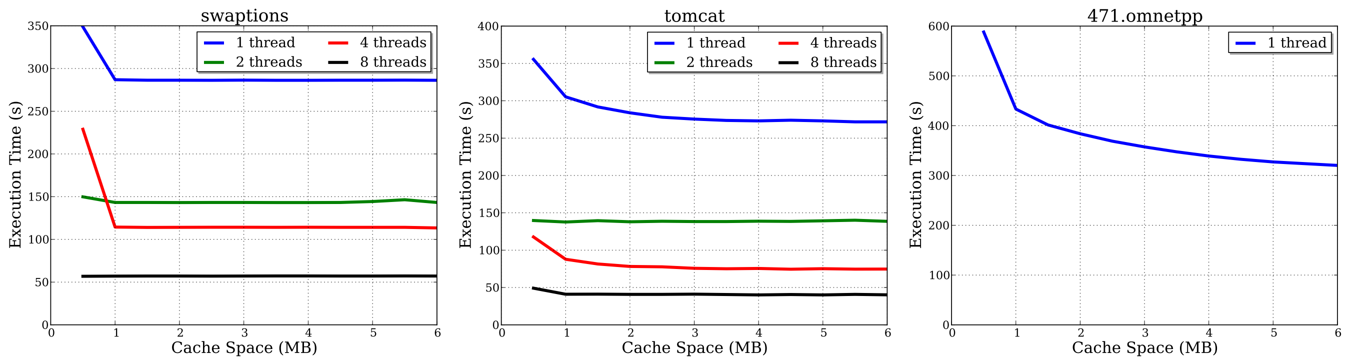


Figure 2: Applications representative of different LLC allocation sensitivities.

threads, and applications that continue to scale up with the number of threads. There are clear differences between suites, with PARSEC clearly being the most scalable.

3.2. Last-Level Cache Sensitivity

We next evaluate how sensitive the benchmarks are to changes in the amount of LLC capacity made available to them. Taking advantage of the way-based cache partitioning mechanism in the LLC, we change the LLC space allocated to a given application from 0.5 MB to 6 MB. In the interests of space, we show only the behavior of three representative applications in Figure 2.

The first conclusion that we can draw is that running an application inside a 0.5 MB direct-mapped LLC is always detrimental. In addition to conflicts from the direct mapping, inclusivity issues with inner levels of cache lead to significant increases in execution time. The second insight is that the LLC is clearly overprovisioned for these applications. We found 44% of the applications only require 1 MB to reach their maximum performance, while 78% of the applications require less than 3 MB. Other authors have observed similar behavior for cloud computing applications [14].

Finally, we did not observe clear *knees* in execution time as we increase the allocated LLC capacity for any application. Previous simulation-based studies took advantage of these knees to propose dynamic cache partitioning techniques. In contrast, performance improves smoothly with the allocated LLC capacity for all applications. The combination of memory-mapping functions, randomized LLC-indexing functions, prefetchers, pseudo-LRU eviction policies, as well as having multiple threads simultaneously accessing the LLC, serve to remove clear working-set knees in the real system.

Next, we classify applications according to their LLC sensitivity, ignoring the pathological direct-mapped 0.5 MB case. Figure 2 shows the behavior of three representative applications. *Low utility* applications yield the same performance despite increased available LLC space (swaptions). *Saturated utility* applications benefit from extra LLC space up to a saturation point (tomcat). Finally, *high utility* applications always benefit from more LLC space (471.omnetpp). We observe that as we increase the number of threads assigned to an application, the LLC sensitivity decreases. Performance is less dependent on LLC size with more cores as there is both a larger aggregate L1 and L2 private cache, and greater overlap of off-chip memory accesses from different cores.

Table 2: Summary of LLC allocation sensitivity.

Suite	Low	Saturated	High
PARSEC	blackscholes, bodytrack, dedup, ferret, fluidanimate, freqmine, raytrace, vips, **streamcluster**, swaptions,	**canneal**, facesim	**x264**
DaCapo	avrora, sunflow	batik, h2, jython, luindex, **tomcat**, **tradesoap**	eclipse, fop, lusearch, pmd, **tradebeans**, **xalan**
SPEC	436.cactusADM, **437.leslie3d**, **450.soplex**, 453.povray, 454.calculix, **459.GemsFDTD**, **462.libquantum**, **470.lbm**	**429.mcf**, 473.astar, 482.sphinx3	**471.omnetpp**
Parallel applications	–	**paradecoder**, stencilprobe	browser animation, **g500**
μbench-marks	–	**ccbench**, **stream uncached**	

Table 2 lists the benchmarks in each suite that belong to each one of the three LLC utility categories. Applications with more than 10 LLC accesses per kilo-instruction are highlighted in bold. These applications may lead to on-chip LLC and off-chip DRAM bandwidth contention and LLC pollution, even if they do not benefit (in terms of execution time) from the allocated space. PARSEC applications have much more relaxed LLC requirements than the other suites. SPEC CPU 2006 applications do not generally benefit from the large LLC, but do have a large number of accesses to the LLC.

3.3. Prefetcher Sensitivity

We next characterize the sensitivity of applications to the behavior of hardware prefetchers, because certain prefetchers are a shared resource that cannot be partitioned (unlike hyperthreads and LLC). In a multi-programmed environment, access streams from different applications could impact sensitive applications if they degrade prefetcher efficacy.

There are four distinct hardware prefetchers on Intel Sandy Bridge platforms: 1) Per-core DCU IP-prefetchers look for sequential load history to determine whether to prefetch the data to the L1 caches; 2) DCU streamer prefetchers detect multiple reads to a single cache line in a certain period of

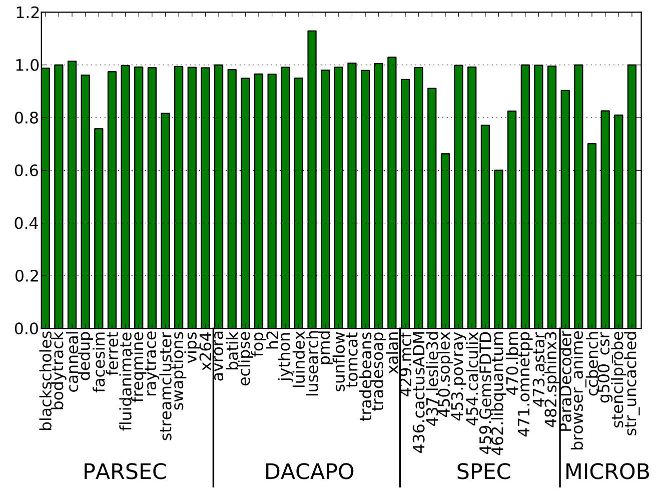


Figure 3: Normalized execution time when enabling all prefetchers enabled w.r.t. all prefetchers disabled.

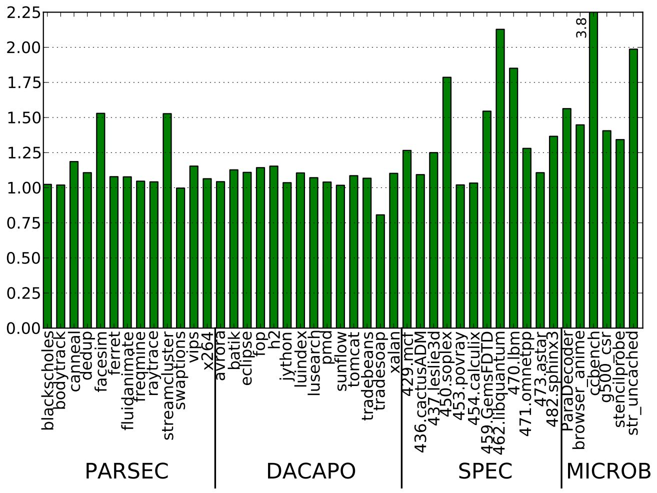


Figure 4: Increase in execution time when running with a bandwidth hog microbenchmark.

time and choose to load the following cache lines to the L1 data caches; 3) Mid-Level cache (MLC) spatial prefetchers detect requests on two successive cache lines and are triggered if the adjacent cache lines are accessed; 4) MLC streaming-prefetchers work similarly to the DCU streamer-prefetchers, which predict the immediate future access patterns based on the current cache line readings. We can activate/deactivate all the DCU and MLC prefetchers by setting the corresponding machine state register (MSR) bits.

Figure 3 shows the reduction in execution time of all the evaluated applications when all prefetchers are active normalized to the configuration with all prefetchers disabled. In general, the evaluated applications are more sensitive to the DCU spatial prefetcher, but there are applications in which the MLC prefetcher is also important for their final performance. Nearly all applications are insensitive to the prefetcher configuration (36 out of 46). In general, we have observed that as we increase the number of threads running in the system, the effectiveness of the prefetcher is reduced as a consequence.

In the case of PARSEC applications, only `facesim` and `streamcluster` benefit from the prefetchers. No DaCapo application benefits from the prefetchers, and `lusearch` performance even degrades when the prefetchers are active. In the case of the additional applications and microbenchmarks, only `ccbench`, `g500` and `stencilprobe` benefit from the prefetcher. In contrast, SPEC benchmarks are more sensitive to the prefetchers, with `450.soplex`, `459.gemsFDTD`, `462.libquantum` and `470.lbm` being the most sensitive applications.

3.4. On-Chip LLC and DRAM Bandwidth Sensitivity

It is important to characterize how sensitive applications are to the amount of bandwidth contention occuring in the system because bandwidth is a shared resouce that cannot be partitioned (unlike hyperthreads and LLC capacity). In a multi-programmed environment, access streams from different applications could cause performance degradation of sensitive applications if they oversubscribe particular network links, memory channels, or MSHR resources.

We characterize applications according to their performance when running together with a bandwidth-hogging microbenchmark (`stream_uncached`), which streams through memory without caching data in the LLC using specially tagged load and store instructions. Bandwidth-sensitive applications will suffer from being run concurrently with this benchmark. Figure 4 shows the increase in execution time of all applications when running with `stream_uncached`. Only two PARSEC applications suffer (`fluidanimante` and `streamcluster`), while DaCapo applications are not affected much by bandwidth contention. In the case of SPEC, some benchmarks are not affected at all (`436.cactusADM`, `453.povray`, `454.calculix`, `473.astar`) and others are heavily affected (`450.soplex`, `459.gemsFDTD`, `462.libquantum`, `470.lbm`). In contrast, all the added parallel applications are bandwidth sensitive. In general, the evaluated applications are more sensitive to bandwidth contention than to the prefetcher.

3.5. Clustering Analysis

Using the characterization of the applications, we select a subset of the benchmarks that are representative of various responses to resource allocations. Following in the footsteps of [27], we use the machine learning technique of hierarchical cluster analysis to select representative benchmarks. Picking representative applications enables us to make application co-scheduling pairings tractable to evaluate. Additionally it allows us to better understand how classes of applications behave when sharing the LLC.

We use a popular hierarchical clustering algorithm [27] provided by the Python library `scipy-cluster` with the single-linkage method because it simultaneously looks at multiple clustering possibilities, allowing the user to select the desired number of clusters using a *dendrogram*.

In order to perform the cluster analysis, we first create a feature vector for each application using the values in the

Table 3: Cluster representatives

Suite	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
PARSEC	–	–	**ferret**	x264	**dedup**	–
DaCapo	h2	–	sunflow	**fop**	avrora	**batik**
SPEC	**429.mcf**	**459.gems-FDTD**	–	–	–	–
μbench-marks	ParaDe-coder	ccbench	–	browser animation	stencil-probe	–
Charac-teristics	Low scalability, sensitive to LLC	Low scalability, prefetcher and bandwidth sensitive	High scalability, reduced LLC utility	Saturated scalability, LLC sensitive	Saturated scalability, LLC insensitive	Saturated scalability, bandwidth sensitive

previous subsection: 1) execution time as we increase the number of threads; 2) execution time as we increase the LLC size; 3) prefetcher sensitivity; and 4) bandwidth sensitivity. All metrics are normalized to the interval $[0,1]$. In total we use vectors with 19 features $(7+10+1+1)$.

The clustering algorithm finds the smallest Euclidean distance of a pair of feature vectors and forms a cluster containing that pair. It continues selecting the next smallest distance between a pair and forms another cluster. Linkage criteria can be uses to adjust cluster formation. The *single-linkage* we selected uses the minimum distance between a pair of objects in different clusters to determine the distance between them.

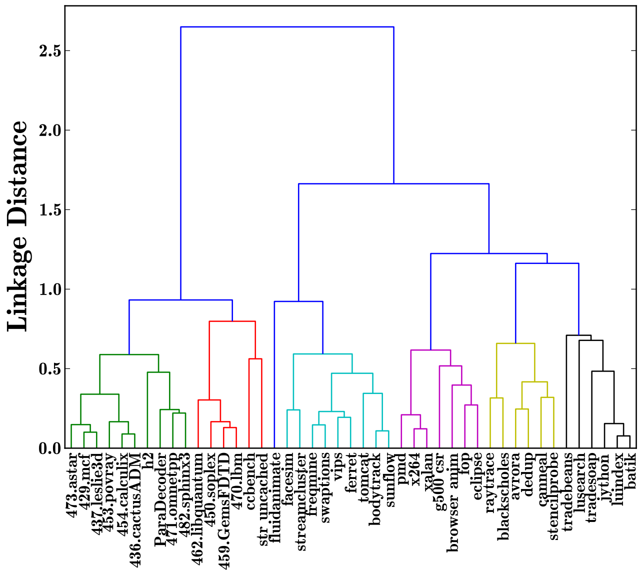


Figure 5: Clustering based on execution time, LLC space, memory bandwidth, and prefetcher sensitivity.

Figure 5 shows the dendogram for the studied applications. The Y-axis represents the linkage-distance between applications. Applications within a distance of 0.9 are set with the same color. On the X-axis, benchmarks are positioned close to each other when the distance metric is smaller. Benchmarks that are outliers have larger linkage distances to the rest of the clusters.

The first two clusters are comprised of applications with low thread scalability. The first cluster is more sensitive to LLC space, but less sensitive to bandwidth and the prefetcher. Applications in the third cluster present high thread scalability and low cache utility and are insensitive to the prefetcher. The last three clusters are comprised of applications with saturated thread scalability, but different cache utility. The fourth cluster is more sensitive to the cache space than the rest, the fifth is insensitive to cache space, and the sixth is insensitive to bandwidth contention. There is also a cluster with only one application (fluidanimate), which stands apart as it only runs correctly when allocated a number of threads that is a power of 2. Due to this irregularity, we do not consider this cluster any further in our analysis.

The representative of each cluster is the benchmark closest to the centroid of the cluster. Table 3 lists the representatives per benchmark suite (when such a representative exists in a cluster). The overall representative is highlighted in bold. From this point onward in the paper we perform a more detailed analysis of the representative applications only.

4. Energy-Performance Tradeoffs

Our next experiments explore the power, energy, and performance tradeoffs available in our system.

Controlling the number of cores assigned to an application, and the frequency at which those cores run, is the most well-studied way by which to change energy consumption. However, it is worth noting that making energy-efficient optimizations sometimes involves counter-intuitive choices. For example, activating additional cores or raising frequency increases power consumption, but can result in lower overall energy consumption per task, since the task may finish earlier and we therefore expend less energy keeping the entire platform active. This operating scenario is often described as *race-to-halt*, meaning that the optimal energy efficiency is obtained by optimizing for the highest performance to allow the platform to shut down and save energy when the task completes. Conversely, a memory-bound application that is allocated additional cores or run at a higher frequency is unlikely to realize any performance benefits, but would consume additional energy as a result of running at a higher power level while waiting for data to be provided by the memory system.

Cache capacity allocation decisions are usually more straightforward, and typically only impact energy by changing the number of LLC misses an application incurs. LLC misses consume energy as the required data must be fetched from DRAM and the runtime of the program is sometimes increased as well. Socket power does not change as a function of the cache size given to an application, since current hardware has no capability to power-gate individual cache subsets. This motivates us to hand underutilized capacity over to another application instead.

To better understand the space of possible performance and energy tradeoffs, we executed each respresentative with all possible thread and way allocations to measure the performance and energy of each resource set. Each benchmark is tested with 1–8 threads and 1–12 cache ways (96 different allo-

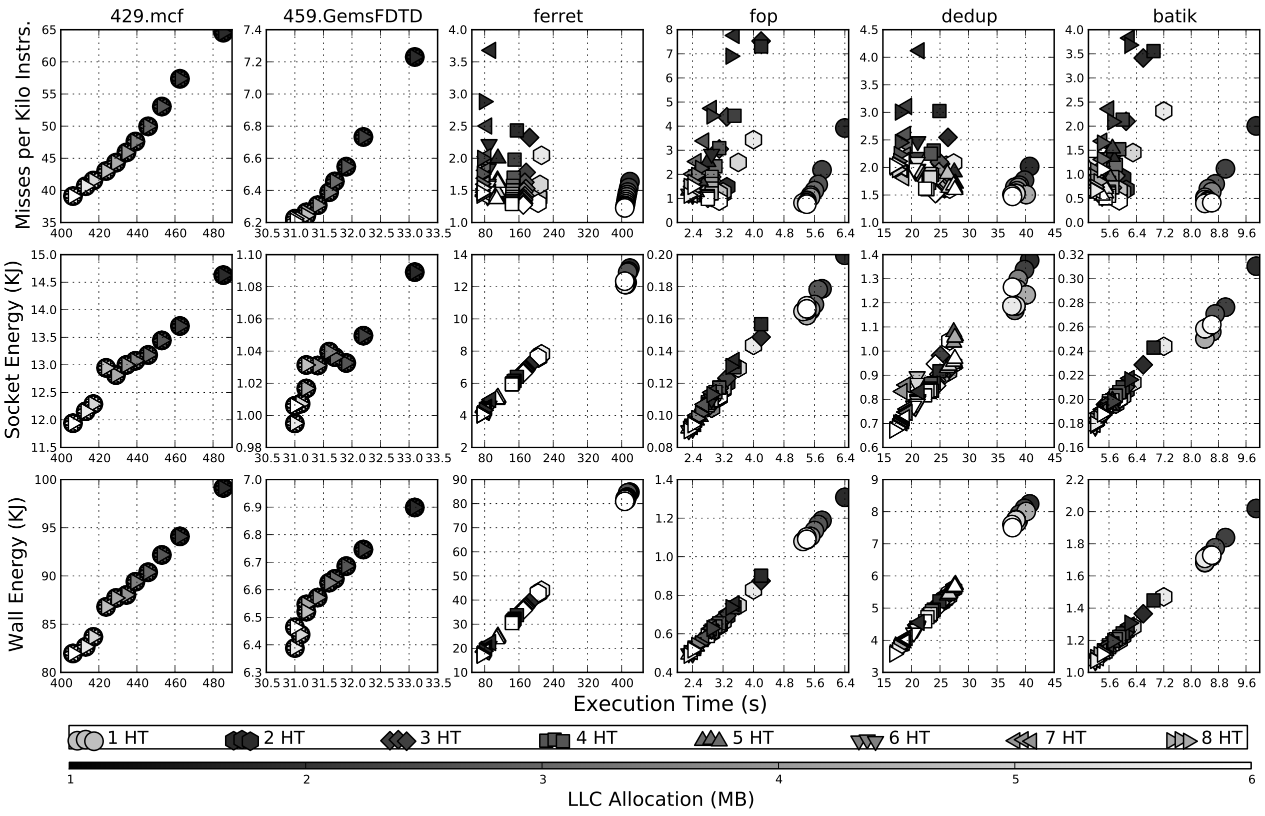


Figure 6: Example performance, energy and miss rate of resource allocations

cations). Figure 6 shows plots of the runtime, LLC misses per kilo instruction (MPKI), and total socket and wall energy consumption of all possible resource allocations for the six cluster representatives. When considering execution time versus miss rate, we can see that some applications have runtimes that are tightly correlated with miss rate (429.mcf and fop), while others are completely independent (ferret and dedup), and some see diminishing returns (459.GemsFDTD and batik). We can also see many points with nearly identical execution time but varying resource allocations, indicating that there should be spare resources available for a background computation to use when the application is running.

When considering energy, our measurements strongly suggest that we are operating in the *race-to-halt* scenario for nearly all of our benchmarks. This is specially significant in the case of the wall energy, since the energy consumed in other parts of the system adds to total energy consumption. While there are a spread of points to consider when picking an allocation that minimizes LLC miss rate at a particular execution time, for nearly all benchmarks this curve narrows significantly when energy is factored in. We see this effect because in general miss rates are correlated with both increased energy and increased execution time, limiting the possibilities for a high-miss-rate allocation that also has lower energy via a faster runtime, or a faster-runtime allocation that expends more energy than it saves.

A important final point to note when looking at Figure 6 is that for a given target runtime or energy there are multiple resource allocations that are equivalent to one another (allowing us to use larger LLC allocations to make up for smaller core allocations or vice versa). Figure 7 illustrates this point by showing the contour plots of the wall energy for each benchmark. Very similar figures are obtained when considering runtime and socket energy. Significantly, many applications have one or more energy-optimal configurations that are not the *largest* allocation. The additional resources contained in the largest allocations did not help these applications improve runtime, and thereby energy. For instance, some applications do not benefit from having more than one assigned thread (429.mcf and 459.GemsFDTD), others require all threads to reach the optimal energy consumption (dedup and ferret), and some applications have a margin of possible thread counts that are still energy-efficient (batik and fop). More importantly, all of them can yield some space in the LLC without affecting their performance, ranging from 0.5 MB (429.mcf) to 4 MB (batik and ferret). This resource gap between equally optimal allocations presents us with an opportunity: we could run additional work concurrently on the remaining LLC and core resources, assuming we can prevent destructive interference.

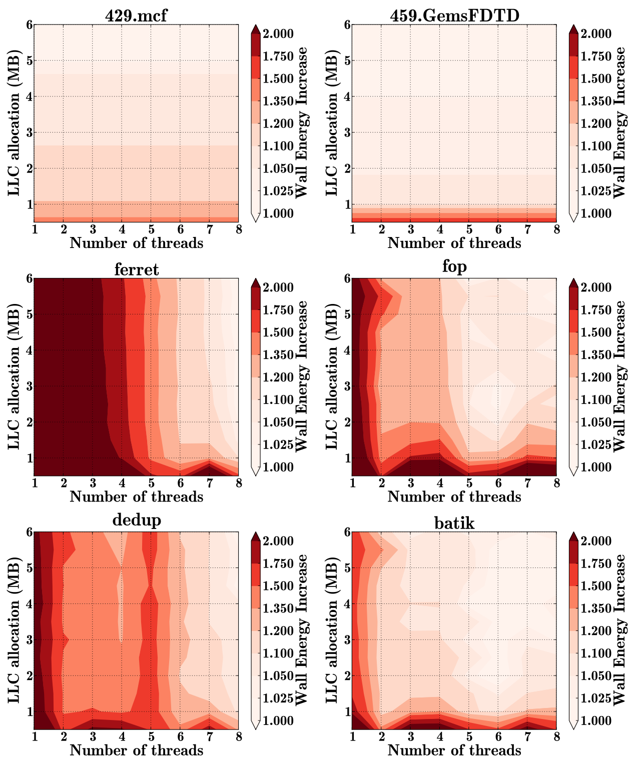


Figure 7: Wall energy contour plots for the cluster representatives. Darker colors represent higher energy consumptions.

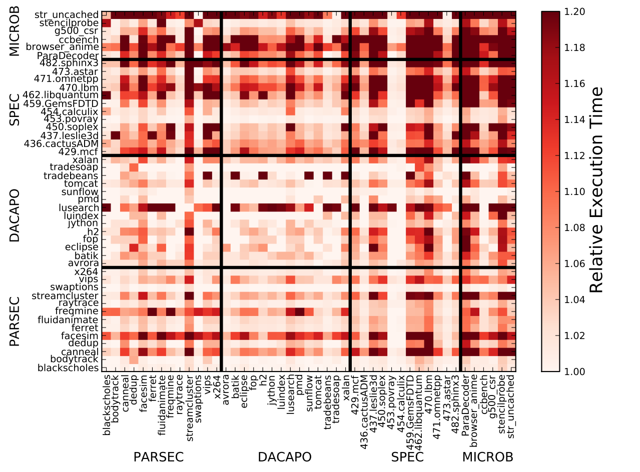


Figure 8: Normalized execution time of the foreground application (X-axis), when run concurrently with a background application (Y-axis).

5. Multiprogram Analyses

In this section, we show that we can often take advantage of the above observation on real hardware and reduce the resource allocation of the foreground application and run a concurrent background application to save energy and improve system throughput, without impacting foreground application performance. For some combinations of applications, we can effectively consolidate applications without even partitioning the LLC. However, other combinations require LLC partitioning to protect the foreground application's performance. We examine the relative effectiveness of various LLC partitioning strategies for these cases. Finally, for some combinations, not even LLC partitioning is effective, and we describe which additional mechanisms would be useful to enable effective consolidation.

5.1. Performance Degradation in a Shared Cache

To begin, we run all possible pairs of applications together with no cache partitioning. We run each multithreaded application on 2 cores with 2 active HTs, making a total of 4 threads per application. In addition to LLC capacity, both applications are sharing the on-chip interconnection network to access the LLC and off-chip memory bandwidth.

Figure 8 shows a heat map of the relative execution time of the foreground application for all possible pairs of applications. The values are normalized to the execution time of the application when running alone on the system with 2 cores with 2 active HTs. For example, canneal's execution time when running with `streamcluster` in the background increases 29% (dark color), while the execution time of streamcluster is affected much less (8.3%, bright color) when running with canneal in the background. It is important to note that these relationships can be asymmetric.

Some applications are not affected when sharing the machine with a background application. This is the case for 22 out of 45 applications, which are affected less than 2.5% on average. The performance of the other applications is more sensitive to contention for the shared hardware resources. SPEC CPU 2006 and the additional parallel applications are generally more contention sensitive, since these benchmarks are more affected when running together with a background application, and also affect the performance of foreground applications from all suites if run in the background. The opposite situation occurs with DaCapo benchmarks, which only slightly affect other foreground applications. The applications that suffer most from sharing (average slowdown over 10%) are canneal, lusearch, 471.omnetpp, ParaDecoder, browser_animation and stream_uncached.

Using the heat map, we can detect aggressor benchmarks that more strongly affect foreground applications' performance. There is one aggressor application in PARSEC (`streamcluster`), none in DaCapo, 5 in SPEC CPU 2006 (`437.leslie`, `450.soplex`, `459.gemsFDTD`, `462.libquantum`, and `470.lbm`), and all the new applications and microbenchmarks except `ccbench`. The average slowdown for the foreground application is over 10% on average when running against these benchmarks.

5.2. Mitigating Degradation with Partitioning

We consider three cache partitioning policies in this section. As evaluated above, the baseline strategy is to allow both applications to share the entire cache (*shared*). The other two strategies are to statically partition the cache between the two apps, in one case splitting the LLC evenly down the middle (*fair*), and in the second case giving each application some uneven allocation (*biased*). We evaluate all possible biased allocations and report results for the one that is best on average, which is always one of the allocations determined to be optimal when the foreground application is run alone.

Figure 9 presents the relative effectiveness of these three strategies at preserving the performance of the cluster-representative applications. We run two applications simultaneously, each with 4 hyperthreads on two cores. The foreground application's performance is measured, while the second application is run continuously in the background. All values are normalized to the execution time of the foreground application running in the shared case. For some foreground applications (C3, C5, C6), less than 5% degradation is present for the shared case, indicating their limited sensitivity. Thus, the improvement provided by partitioning is minor. For some applications pairs, degradation is somewhat or completely mitigated by biased partitioning, but not mitigated by fair partitioning, indicating that small allocations damage performance, while large allocations provide sufficient capacity and protection. For some applications pairs, degradation is not mitigated by any partitioning, indicating that the applications are contending for shared resources other than cache capacity.

The fact that degradation remains present in some cases, even though we have sequestered the applications on disjoint sets of cores and (in the biased case) provided them with optimally-sized cache allocations, implies that bandwidth contention on the on-chip ring interconnect or off-chip DRAM interface is to blame. Prior work [23] has proposed mechanisms to partition such bandwidth resources, but unfortunately they are not available on extant hardware. While there is little we can do to mitigate this contention where it exists on our platform, there are a number of points where energy optimization through consolidation is still possible.

5.3. Energy Savings of Consolidation

Assuming degradation of the foreground application can be managed acceptably, and assuming the foreground application can afford to lose some resources, there is an opportunity to save energy. Figure 10 shows the possible energy savings as measured by running each of the applications once, consolidated with different partitioning strategies, as compared to running the applications once in a time-multiplexed way. This plot uses the optimal allocation of cores and cache capacity for each benchmark pair and scheduling policy.

Moderate energy savings are possible, especially when considering the fact that the best possible reduction over sequentially running two applications with equal running time is 50%.

However, note that the specific partitioning strategy does not have a significant impact for most application pairs. Naive sharing suffices in most cases. The primary exception is `GEMS-FDTD` running with `fop`, `dedup` and `batik`, and even in this case the fair split performs as well as the best biased allocation.

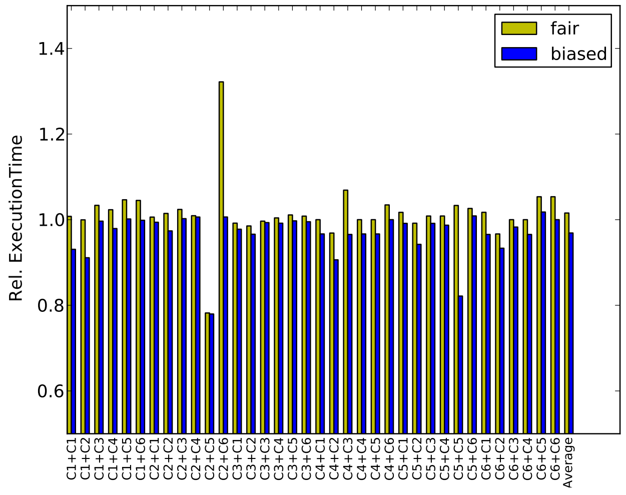


Figure 9: Effect of different partitioning strategies on a foreground application in the presence of different, continuously-running background applications.

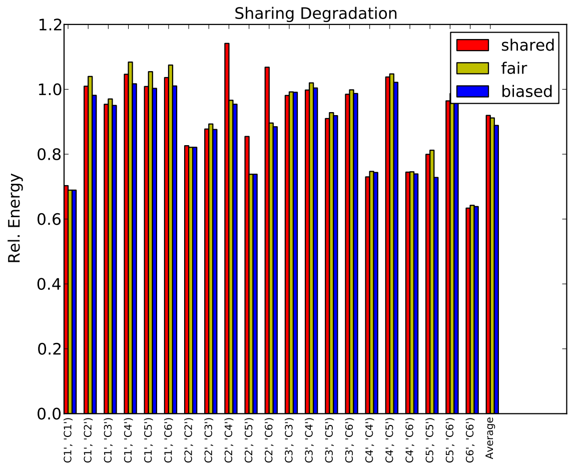


Figure 10: Socket and wall energy values when running with a shared, evenly and optimally partitioned LLC normalized to the applications running sequentially on the whole machine

Figure 11 shows the weighted speedup of the application pairs running together with different partitioning strategies as compared to each running alone. The speedup results indicate that many schedules save energy through spatial multiplexing even though the applications themselves are slowed down.

We have seen the potential for consolidation, but found that the ideal static partitioning is generally not far from either the naive fair partitioning case or the naive shared case. Even an oracular static capacity allocator seems ineffective at squeezing in extra background utilization. However, such an optimal static allocation still cannot take into account phase-based behavior of the applications. We now present our implementation

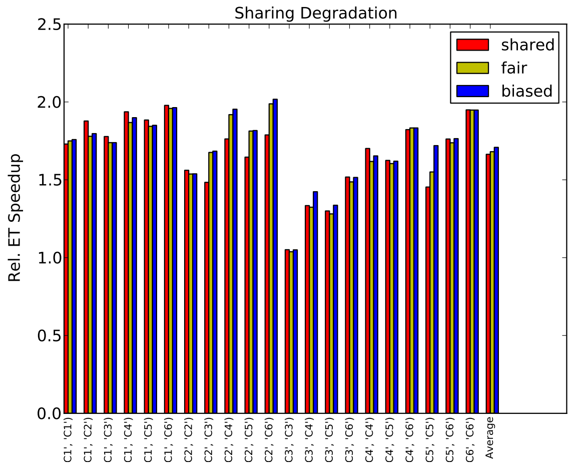


Figure 11: Relative performance of different partitioning strategies normalized to applications running sequentially on the whole machine.

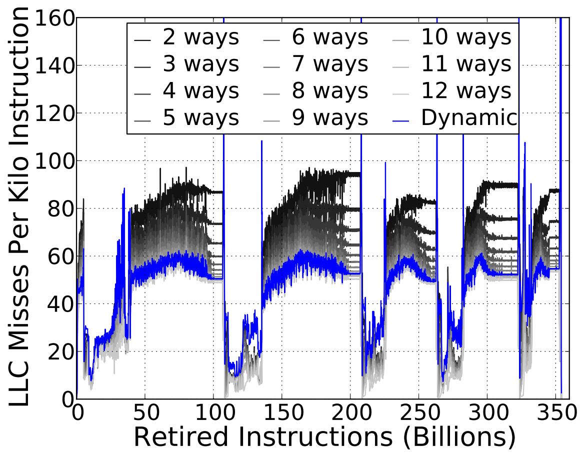


Figure 12: `429.mcf` LLC MPKI phase changes with different static and dynamic LLC allocations.

a utility-based policy for controlling a dynamic cache capacity allocation framework to further maximize consolidation effectiveness while mitigating performance penalties.

5.4. Dynamic Cache Reallocation for Background Application Throughput

In the previous section we focused on the energy saved in a scenario where we have a finite set of tasks to complete as quickly as possible, such as might be encountered on a mobile device. However, in a cloud computing environment, the significant capital investment in datacenter hardware leads cloud providers to attempt to utilize all machines as much as possible. Thus, the objective is to maximize the throughput of background tasks on underutilized hardware without damaging foreground application performance.

Applications often have phased behavior with very different resource requirements during each phase. Figure 12 shows the number of LLC misses per kilo-instruction under different cache allocations for `429.mcf`. This application has five phases with low and high LLC MPKIs. In phases with high LLC MPKI, the LLC allocation has a significant impact in the performance of the application. In these phases, `429.mcf` requires 4.5 MB (9 ways) to reach 95% of its maximum performance, while in the other phases, only 1.5 MB (3 ways) are required. In general, applications present a wide range in the number of phases and the LLC sensitivity of these phases.

Algorithm 5.1: PHASE DETECTION ALGORITHM()

```
if not new_phase {
  if (|avg_MPKI-current_MPKI| >MPKI_THR_1) {
    new_phase=1;
    return 2;
  }
}
else if (|avg_MPKI-current_MPKI| <MPKI_THR_2)
  new_phase = 0;
return new_phase;
```

Algorithm 5.2: DYNAMIC CACHE PARTITIONING ALGORITHM()

```
if (phase_det()==2) {
  phase_starts=1;
  set_cache_to_6MB(fg)
}
else if (phase_det()==0 and phase_starts==1) {
  if (|last_MPKI-current_MPKI | < MPKI_THR_3) {
    if (cache_allocated > 1MB)
      allocate_less_cache(fg)
    else phase_starts=0; /* Keep 1MB */
  }
  else {
    if (cache_allocated < 6MB)
      allocate_more_cache(fg) /* Keep previous allocation */
    phase_starts=0;
  }
}
last_MPKI = current_MPKI;
```

Furthermore, while we have demonstrated a methodology for determining what static cache capacity allocations preserve performance, doing this offline analysis for all possible foreground applications is unlikely to be feasible in practice. These two facts motivate the use of a dynamic mechanism that changes resource allocations according to these requirements. We now evaluate such a mechanism in terms of its ability to provide improved background application throughput over a static allocation scheme.

To enable a dynamic mechanism that adapts to the requirements of the foreground application, we have created a software framework to monitor the behavior of the foreground application and respond to phase changes within that application. Since the only resource that we can directly control at runtime is the cache capacity allocation, the metric of most relevance is the number of LLC misses. Consequently, the framework detects phase changes by examining the change in LLC misses per kilo-instruction and reacts to changes in phases by readjusting the cache allocation. Algorithm 5.1 shows the pseudocode of this phase detection mechanism. Phase changes are detected every 0.1 seconds.

Specifically, when the foreground application begins running, or whenever the framework detects a phase change, the framework gives the foreground application as much cache as possible. Then, the framework gradually reduces the fore-

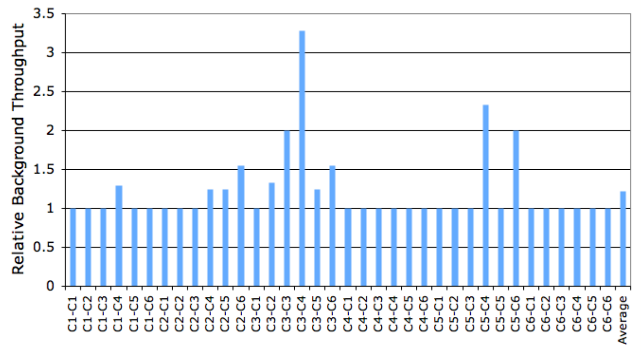


Figure 13: Summary of improvements to rate of background task completion compared to the static cache allocation with optimal foreground application performance.

ground LLC allocation until a bad performance effect is observed (misses go up). The background application(s) are given the remaining resources. The framework directly controls the cache allocation of each application by making use of specific hardware mechanisms to partition the LLC at way granularity. On a reallocation, there is no flushing of data since the partitioning mechanism only affects future evicted cache lines [32]. Algorithm 5.2 shows the pseudocode for this cache partitioning mechanism.

We must set the thresholds for MPKI derivatives correctly to detect phases and too-small allocations. For applications with a rapid flutter of MPKI (relative to the time it takes to slowly reduce allocation size) the final LLC allocation might be unnecessarily large. Data remaining unevicted in deallocated ways might let allocation size shrink too much, since the performance effects of the shrinkage are masked. In our case, we performed a sensitivity study that lead to the selected parameters: $MPKI\_THR_1 = 0.02$, $MPKI\_THR_2 = 0.02$, and $MPKI\_THR_3 = 0.05$, but found that the below results are largely insensitive to small parameter changes.

Across our benchmarks, we find that the framework is able to achieve within 1% of the best performance of the best static allocation. Given that the performance is a good match to the best static allocation, we must determine how the extra cache capacity made available by the dynamic mechanism to the background tasks during phases of the foreground application with low cache utility improves background task bandwidth. Figure 13 shows that in some cases significant throughput increases (22% on average) are made by this resource availability. These bandwidth improvements are relative to the smallest static allocation that is within 1% of overall optimal performance. However, for many cases, the limited number of phases in the foreground application or its high sensitivity to LLC allocation size does not allow for additional improvements from deploying the dynamic mechanism as compared to a near-optimal static policy. However, in the right context, the dyanmic partitioning mechanism can make application consolidation onto hardware with a shared LLC far more effective.

6. Related Work

Several groups have studied the impact of workload consolidation in shared caches [35, 36, 38] for datacenter applications. They do not explore hardware solutions such as cache partitioning or client workloads.

Several authors have evaluated way-partitioned caches with multiprogrammed workloads [16, 20, 28, 32] including using partitioning to provide a minimum performance to applications [20, 25]. However, these proposals were all evaluated on a simulator, and all but [20] used sequential applications — leading to different conclusions.

Other authors make use of *page coloring* to partition the LLC by sets [10, 24, 33]. Cho and Lin [10] experiment with page coloring on a simulator to reduce access latency in distributed caches. Tam et al. [33] map pages into each core's private color using static decisions. They implement it using Linux and POWER5. Lin et al. [24] evaluate a dynamic scheme using page coloring on an Intel Xeon 5160 processor. However, there is a huge performance overhead to change the color of a page. Another challenge is that the number of available partitions is a function of page size, so increasing page size can make page coloring ineffective or impossible [10, 24, 33]. The experiments on real machines use only 2 running threads and make decisions only every several seconds. In contrast, our approach can change LLC partitions much faster and with minimal performance overhead.

Others have proposed hardware support to obtain the miss rate, IPC curves or hardware utilization information [8, 25, 28] to provide QoS. These approaches require hardware modification and will not work on current processors. Chanda et al. [9] explored on-line methods to extrapolate these curves, but their approach has significant complexity and overhead. Tam et al. [34] use of performance counters on POWER5 to predict miss curves for different page coloring assignments. They use a stack simulator and an LLC address trace to obtain the miss curve and statically assign colors. Others have measured LLC misses to identify thrashing applications [37] or predict contention [13] enabling the scheduler to pick co-running applications with more affinity. Another approach uses hardware thread priorities to rate limit threads [7]. However, the approach only works for SMT processors and non-memory-bound applications. LLC misses can also be used to change the replacement policy of the LLC to partition the cache at finer granularity [29].

7. Conclusion

In this study we found that modern hardware and applications offer possibilities to increase energy efficiency through consolidation for foreground and background applications. By measuring performance and energy on a current microprocessor with experimental support for cache partitioning, we were able to cluster several benchmark suites based on their resource utilization. Unlike previous studies, we did not observe performance knees as we increase the allocated LLC capacity.

We found that, for all classes of applications, pinning threads to cores and sharing the LLC provides speedups and energy savings over running applications alone in sequence, and in many cases is effective at preserving foreground application performance. The 6 MB LLC is typically enough for two applications to share without degradation, irrespective of cache partitioning policy. For 16% of the cases, (almost exclusively those running with a low scalability, sensitive application), uneven cache partitioning can provide significant speedups, 20% on average. We implemented a simple dynamic mechanism that achieves performance within 1% of the optimal uneven allocation. Overall, we find that while naive LLC sharing can often be effective, the combination of a cache replacement mechanism that enables partitioning along with a simple adjustment algorithm can provide significant performance and energy improvements in certain cases. We expect the effectiveness of our dynamic partitioning technique to increase as workloads become more parallel and multiprogrammed, but for it to be less effective on machines with larger cache hierarchies or without support for partitioning other shared hardware resources.

References

[1] Apple Inc. iOS App Programming Guide. http://developer.apple.com/library/ios/DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[4] S. Beamer, K. Asanovic, and D. A. Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley, Nov 2011.

[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[7] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. In *ISCA*, pages 415–426, 2008.

[8] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Trans. Computers*, 55(7):785–799, 2006.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, pages 340 – 351, 2005.

[10] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO*, pages 455–468, 2006.

[11] S. Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *Ottawa Linux Symposium*, page 269–288, 2006.

[12] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Commun. ACM*, 55(7):105–114, July 2012.

[13] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, 2010.

[14] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48, 2012.

[15] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Workshop on Programming Languages and Operating Systems*, pages 1–5, 2011.

[16] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[18] Intel Corp. Intel 64 and ia-32 architectures optimization reference manual, June 2011.

[19] Intel Corp. Intel 64 and ia-32 architectures software developer's manual, March 2012.

[20] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[21] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation – a pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites. Technical report, VSSAD, Intel Corporation, 2007.

[22] S. Kamil. Stencil probe, 2012. http://www.cs.berkeley.edu/~skamil/projects/stencilprobe/.

[23] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 89–100, Washington, DC, USA, 2008. IEEE Computer Society.

[24] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367 –378, feb. 2008.

[25] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[26] Perfmon2 webpage. perfmon2.sourceforge.net/.

[27] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA*, pages 412–423, 2007.

[28] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, pages 423–432, 2006.

[29] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *ISCA)*, June 2011.

[30] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity*, 2009.

[31] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark suite. http://www.spec.org.

[32] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *HPCA*, pages 117–128, 2002.

[33] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA*, 2007.

[34] D. K. Tam, R. Azimi, L. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS*, pages 121–132, 2009.

[35] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011.

[36] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *ISPASS*, pages 2–11, 2011.

[37] Y. Xie and G. H. Loh. Scalable shared-cache management by containing thrashing workloads. In *HiPEAC*, pages 262–276, 2010.

[38] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP*, pages 203–212, 2010.