CloudCache: Expanding and Shrinking Private Caches

Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers
Computer Science Department, University of Pittsburgh
{abraham,cho,Childers}@cs.pitt.edu

Abstract

The number of cores in a single chip multiprocessor is expected to grow in coming years. Likewise, aggregate on-chip cache capacity is increasing fast and its effective utilization is becoming ever more important. Furthermore, available cores are expected to be underutilized due to the power wall and highly heterogeneous future workloads. This trend makes existing L2 cache management techniques less effective for two problems: increased capacity interference between working cores and longer L2 access latency. We propose a novel scalable cache management framework called CloudCache that creates dynamically expanding and shrinking L2 caches for working threads with fine-grained hardware monitoring and control. The key architectural components of CloudCache are L2 cache chaining, inter- and intra-bank cache partitioning, and a performance-optimized coherence protocol. Our extensive experimental evaluation demonstrates that CloudCache significantly improves performance of a wide range of workloads when all or a subset of cores are occupied.

1. Introduction

Many-core chip multiprocessors (CMPs) are near—major processor vendors already ship CMPs with four to twelve cores and have roadmaps to hundreds of cores [1, 2]. Some manufacturers even produce many-core chips today, such as Tilera's 100-core CMP [3] and Cisco's CRS-1 with 192 Tensilica cores [4]. For current and future CMPs, tile-based architectures are the most viable. A tile-based CMP is comprised of multiple identical tiles each with a compute core, L1/L2 caches, and a network router. In this kind of design, the tile organization is not dramatically changed successive processor generations. This trend implies that more tiles will lead to more aggregate L2 cache capacity.

Effectively managing a large L2 cache in a many-core CMP has three critical challenges: how to manage capacity (cache partitioning), how to avoid inter-thread interference (performance isolation), and how to place data (minimizing access latency). These challenges are more acute at a large core count, and current approaches for a small number of cores are insufficient. A shared cache suffers from uncontrolled capacity interference and increased average data access latency. A private cache does not utilize total L2 cache capacity efficiently. Although many hybrid L2 cache management techniques try to overcome the deficiencies of shared and private caches [5–11], their applicability to many-core CMPs at scale is uncertain.

This work was supported in part by NSF grants CCF-0811295, CCF-0811352, CCF-0702236, CCF-0952273, and CCF-1059283.

While much effort is paid on how to program and utilize the parallelism from future CMPs, the accelerating trend of extreme system integration, clearly exemplified by data center servers with cloud computing, will make future workloads more heterogeneous and dynamic. Specifically, a cloud computing environment will have many-core CMPs that execute applications (and virtual machines) which belong to different clients. Moreover, the average processor usage of a data center server is reportedly around 15–30%. However, peak-time usage often faces a shortage of computing resources [12, 13]. These characteristics and the need to run heterogeneous workloads will become more pronounced in the near future, even for desktops and laptops.

Future heterogeneous workloads will need scalable and malleable L2 cache management given the hundreds of cores likely in a CMP. Scalability must become the primary design consideration. Moreover, a new cache management scheme must consider both low and high CPU utilization situations. With low utilization, the excess L2 cache capacity in idle cores should be opportunistically used. Even when all cores are busy, the cache may still be underutilized and could be effectively shared.

This paper proposes *CloudCache*, a novel distributed L2 cache substrate for many-core CMPs. CloudCache has three main components: *dynamic global partitioning*, *distance-aware data placement*, and *limited target broadcast*. Dynamic global partitioning tries to minimize detrimental cache capacity interference with information about each thread's capacity usage. Distance-aware data placement tackles the large NUCA effect on a switched network. Finally, limited target broadcast aims to quickly locate a locally missing cache block by simultaneously inspecting nearby non-local cache banks. This broadcast is limited by the distance-aware data placement algorithm. Effectively, CloudCache overcomes the latency overheads of accessing the on-chip directory. Our main contributions are:

- **Dynamic global partitioning.** We introduce and explore distributed dynamic global partitioning. CloudCache coordinates bank and way-level capacity partitions based on cache utilization. We find that dynamic global partitioning is especially beneficial for highly heterogeneous workloads (e.g., cloud computing).

Scheme	Org.	Type	Key idea	Dynamic Partition	Explicit alloc.	Dist. aware.	Tiled CMP	QoS	Coherence
CMP-DNUCA [14]	Dist.	S	Private data migration			✓			Dir
VR [5]	Dist.	S	Victim replication			✓	✓		Dir
CMP-NuRAPID [6]	Dist.	P	Decoupled tag			✓	✓		BC
CMP-SNUCA [15]	Dist.	S	Dynamic sharing degree	✓		✓			Dir
CC [7]	Dist.	P	Selective copy						Dir
ASR [8]	Dist.	P	Selective copy w/ cost estimation						BC
UMON [16]	One	S	Utility-based partitioning	✓					BC
V-Hierarchy [17]	Dist.	S	Partitioning for VMs				✓	✓	Dir
VPC [18]	One	S	Bandwidth management					✓	BC
DSR [9]	Dist.	P	Spill, receive					✓	BC
R-NUCA [10]	Dist.	S	Placement w/ P-table			✓	✓		Dir
BDCP [11]	Dist.	P	Bank-aware partitioning	✓	✓	✓			BC
StimulusCache [19]	Dist.	P	Dynamic sharing of excess caches				✓	✓	Dir
Elastic CC [20]	Dist.	P	Local bank partitioning w/ global sharing	✓			✓		Dir
CloudCache	Dist	P	Distance-aware global partitioning	✓	✓	✓	✓	✓	Dir+BC

Table 1. Related cache management proposals and CloudCache. **Organization**: "One" (one logical bank) or "Dist." (distributed banks). **Type**: "S" (shared) or "P" (private). **Dynamic partitioning**: cache capacity can be dynamically allocated. **Explicit allocation**: non-shared cache capacity is explicitly allocated. **Tiled CMP**: applicability to tiled CMP (even if the original proposal was not for tiled CMP). **QoS**: quality of service support. **Coherence**: "BC" (broadcasting-based) or "Dir" (directory-based).

- **Distance-aware data placement and limited target broadcast.** We show the benefit of distance-aware capacity allocation in CloudCache; it is particularly useful for many-core CMPs with a noticeable NUCA effect. The full benefit of distance-aware data placement is realized with limited target broadcast. The performance improvement is up to 16% over no broadcast.
- **CloudCache design.** We detail an efficient CloudCache design encompassing our techniques. The key architectural components are: L2 cache chaining, inter- and intra-bank cache partitioning, and a performance-correctness decoupled coherence protocol.
- **An evaluation of CloudCache.** We comprehensively evaluate our proposed architecture and techniques. We compare CloudCache to a shared cache, a private cache, and two relevant state-of-the-art proposals, Dynamic Spill-Receive (DSR) [9] and Elastic Cooperative Caching (ECC) [20]. We examine various workloads for 16- and 64-core CMP configurations. CloudCache consistently boosts performance of co-scheduled programs by 7.5%–18.5% on average (up to 34% gain). It outperforms both DSR and ECC.

In the remainder of this paper, we first summarize related work in Section 2. Section 3 presents a detailed description of CloudCache and its hardware support. Section 4 gives our experimental setup and results. The paper's conclusions are summarized in Section 5.

2. Related Work

Much work has been done to improve and/or solve the deficiencies of the common shared and private cache schemes. While there are many cache management schemes available, Table 1 summarizes the key ideas and capabilities among the schemes most related to CloudCache. The table compares the schemes according to six parameters. Compared with other techniques, CloudCache (the last row) has notable differences in the context of supporting many-core CMPs: dynamic partitioning that involves many caches, explicit, non-shared cache allocation to each program, awareness of distance to cached data, and quality of service (QoS) support.

CMP-DNUCA [14], victim replication [5], and CMP-NuRAPID [6] place private or read-only data in local banks to reduce access latency. CMP-SNUCA [15] allows each thread to have different shared cache capacity. Cooperative Caching (CC) [7] and Adaptive Selective Replication (ASR) [8] selectively evict or replicate data blocks such that effective capacity can be increased. The utility monitor (UMON) [16] allocates the capacity of a single L2 cache based on utilization. Marty and Hill proposed the Virtual Hierarchy (VH) [17] to minimize data access latency of a distributed shared cache with a two-level cache coherency mechanism. The Virtual Private Cache (VPC) [18] uses a hardware arbiter to allocate cache resources exclusively to each core in a shared cache. These proposals do not support explicit cache partitioning (i.e., capacity interference cannot be avoided), or they are unable to efficiently and dynamically allocate the distributed cache resources.

More recently, Dynamic Spill-Receive (DSR) [9] supports capacity borrowing based on a private cache design. R-NUCA [10] differentiates instruction, private data and shared data and places them in a specialized manner at page granularity with OS support. BDCP [11] explicitly allocates cache capacity to threads with local banks and center banks. It avoids excessive replication of shared data and places private data in local L2 banks. StimulusCache [19] introduced techniques to utilize "excess caches" when some cores are disabled to improve the chip yield. Lastly, Elastic Cooperative Caching (ECC) [20] uses a distributed coherence engine

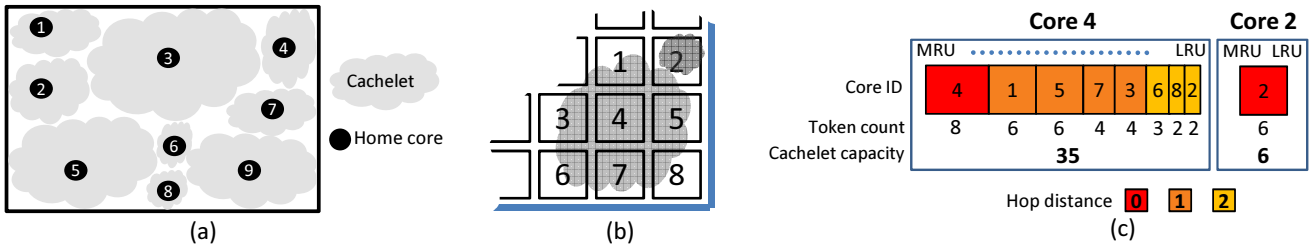

Figure 1. (a) Overview of CloudCache with nine home cores. (b) An example of two home cores; core 4 has a larger cachelet than core 2. (c) An example virtual private L2 cache description of (b). "Core ID" refers to the list of cores contributing to a cachelet. Core IDs are sorted in increasing distance from the home core. "Token count" is the number of cache capacity units contributed to the cachelet. "cachelet capacity" is the sum of all token counts.

for scalability. It allows sharing of the "local partition" of each core if the core does not require all the capacity of its local partition. None of these recent proposals can avoid capacity interference and long access latency at scale.

Compared to these proposals, CloudCache does more effective globally-coordinated dynamic partitioning. Each thread has non-shared exclusive cache capacity, which inherently avoids capacity interference. It also addresses the NUCA problem for a large CMP, caused by distributed cache banks, directory, and memory controllers.

3. CloudCache

We begin with a high-level description of CloudCache. Figure 1(a) depicts nine "home cores" where nine active threads are executed. Home cores have a virtual private L2 cache (which we call a "cachelet") that combines cache capacity from a thread's home core and neighboring cores. While cache banks might be shared among different cachelets, each core is given its own exclusive cachelet to avoid interference with other cachelets. The capacity of a cachelet is dynamically allocated based on the varying demand of the thread on the home core and the demands of threads on neighboring cores (which have their own cachelets). For example, in Figure 1(a), core 3 has been given the largest cache capacity. If core 6 needs to grow its cachelet, the adjacent cachelets (cachelet 3, 5, 8, and 9) adjust their size to give some capacity to core 6. Cachelets are naturally formed in a cluster to minimize the average access latency to data.

Cachelets can be compactly represented. Figure 1(b) gives a second example with only two home cores. Core 4 has a larger cachelet than core 2. Figure 1(c) further shows the LRU stack for core 4 and core 2's cache cachelets. The stack incorporates the cache slices of all neighbor cores that participate in a cachelet. The stack is formed based on the hop distance to a neighbor. The highest priority position (MRU) is the local slice. In core 4, the MRU position has an 8 in this example. The value in a position indicates how many ways out of cache slice are allocated to the thread. The 8 in this case specifies that all 8 ways of the local cache slice have been allocated to the thread on core 4. The next several positions record the capacity from cores that are one hop away (core 1, 5, 7, and 3). These cores provide a capacity of 20 to the thread on core 4. The final positions in the stack are the farthest away (core 6, 8, and 2); they dedicate an additional aggregate capacity of 7. The figure also shows core 2. The thread on this core needs a capacity of 6, which can be provided locally. Lastly, the cores in the core ID list form a "virtual L2 cache chain," somewhat similar to [19]. For example, when core 4 has a miss, the access is directed to core 1, then to core 5, and so on (from the MRU position to later positions).

3.1. Dynamic global partitioning

The allocation of cachelets requires careful global coordination because cache capacity and proximity have to be considered simultaneously to achieve a good decision. CloudCache has a *global capacity allocator* (GCA) for this purpose. The GCA collects information about cache demand changes of home cores and performs global cache partitioning. It uses a utility monitor similar to UMON [16], with an important modification to support many-core CMPs. The original UMON scheme evaluates all possible partition choices with duplicated tags in a set-associative array. In UMON, the number of ways for the duplicated tag array is the number of cores in the CMP multiplied by the associativity of a cache slice. For a many-core CMP, the overhead of the duplicated tag array is high. The original UMON scheme requires a 512-way duplicated tag array per tile for a 64-core CMP with an 8-way L2 cache per tile. To overcome this overhead, we limit the monitoring scope and evaluate each core's additional cache capacity benefit of up to 32 ways, which is four times the local cache capacity for an 8-way slice. For example, a thread with a capacity of 64 ways is able to have at most 96 ways at the next capacity allocation. Our evaluation shows this modification works well with lower hardware cost than the full UMON scheme.[1]

To gather information for capacity allocation decisions, each core sends hit count information to the GCA once every monitoring period. We experimentally determine that 64M cycles works well for our benchmarks. The hit count infor-

[1] In general, this "monitoring range" is a design-time decision based on cache capacity and target workload.

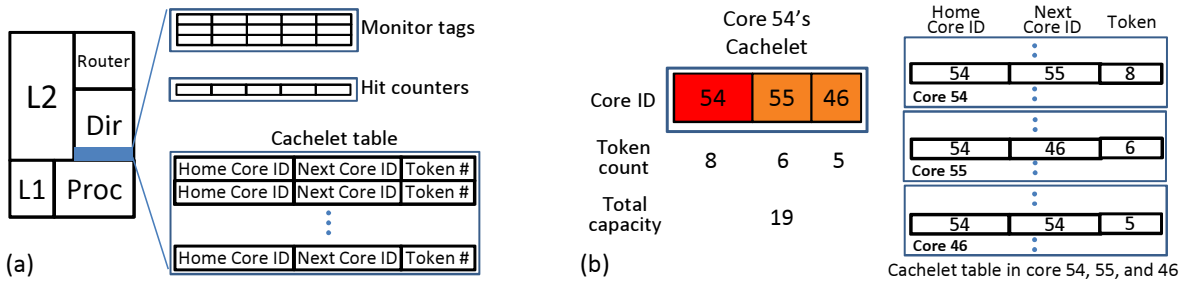

Figure 2. (a) Hardware architecture support for CloudCache. (b) Virtual L2 cache chain example.

mation includes the L2 cache and monitoring tag hit count for each LRU stack position. The network traffic for transmission of this information is very small. Once the GCA receives the counter values from all home cores, the counters are saved in a buffer. The total number of counters is $N \times (K + 32)$ where N is the number of cores and K is the L2 cache associativity. $N \times K$ counters are used for the hit counts and $N \times 32$ tag hit counters are used to estimate the benefit from additional cache capacity up to 32 ways. For example, a 64-core CMP with an 8-way L2 cache slice has 2,560 16-bits counters in a small 5KB buffer. The GCA uses the counter buffer to derive near optimal capacity allocation.

Figure 2(a) shows the per tile hardware architecture for CloudCache. Each tile has monitor tags, hit counters, and a "cachelet table". CloudCache monitors cache capacity usage for each core with the hit counters. The potential benefit of increasing capacity is estimated with the monitor tags. Whenever a cachelet evicts a data block, the address of the evicted data block is sent to the home core so that it can be used to estimate hit counts if the capacity should be increased. The cachelet table describes a virtual private L2 cache as a linked list of the cache slices that form the cachelet. It is used to determine the data migration path on cache evictions and to determine how much of a particular cache slice can be used by a cachelet. Each entry in the cachelet table has three fields: the home core ID, the next core ID, and the token count. The home core ID indicates the owner of the cachelet. When data is found in a particular cache slice, that slice delivers it to the home core. The next core ID indicates the target of an eviction from a cache slice. If the next core ID and the home core ID are the same, then the evicted data is sent to main memory (i.e., next core ID==home core ID is the list tail). The token count indicates how many ways of a cache slice are dedicated to a cachelet's owner core. If this value is '0', then the table entry is invalid.

Figure 2(b) shows an example of the cachelet table. Suppose core 54 needs capacity of 19 ways and this capacity comes from cores 54, 55, and 46. In core 54's cachelet table, the next core ID points to core 55, which provides the next LRU stack of the cachelet. Core 55's cachelet table has an entry for core 54 with a token count of 6 (core 55 may also have its own entry, if it is running a thread—this is not shown). It also has the next core ID, core 46, which points to the last LRU stack of the cachelet. Finally, core 46 has a table entry for core 54 with the next core ID set to 54, the home core for the cachelet. This denotes that this core 46's cache slice is the last LRU stack position in the cachelet.

3.2. Distance-aware data placement

The GCA uses the modified UMON scheme to determine the capacity demand for each thread on the CMP. With this information, the GCA decides which L2 cache(s) to use for a cachelet. It then uses a greedy distance-aware placement strategy on a cachelet for each thread. Cache capacity for each thread is allocated to the local L2 bank first to minimized access latency. If more capacity than L2 bank is allocated to a thread, remote L2 banks should be used for the extra capacity allocation. Our strategy allocates capacity to threads in the order of larger capacity demand. Target L2 banks with shorter distance to the thread are selected.

Once cache banks are selected for threads, chain link allocation is performed. The local L2 bank (i.e., the closest L2 bank to the thread) is located in the top LRU stack of the chain link. The farthest L2 bank is used for the bottom LRU stack, and is connected to the main memory.

3.3. Fast data access with limited target broadcast

CloudCache quickly locates nearby data on a local cache miss with *limited target broadcast*. This technique effectively hides directory lookup latency. In a packet-based network, the directory manages the order of requests such that packets avoid race conditions. To access a remote L2 cache, a core needs to access the directory first even if the remote L2 cache is only one hop away. To avoid this inefficiency, we design a *limited target broadcast protocol* (LTBP). LTBP allows fast access to private data while shared data is processed by a conventional directory protocol. To reduce network traffic, LTBP sends broadcast requests only to remote L2 cache slices that are allocated to the home core.

LTBP consists of two parts for the directory and L2 cache. LTBP for the directory processes a request for private data from a non-owner core. When the directory receives a non-owner request, it sends a broadcast lock request to the owner cache. If the owner cache accepts the broadcast lock request, the directory processes the non-owner's request. When the data block is locked for broadcast, the owner cache does not respond to a broadcast request for the data block. In

Core's pipeline	Intel's ATOM-like two-issue in-order pipeline with 16 stages at 4GHz
Branch predictor	Hybrid branch predictor (4K-entry gshare, 4K-entry per-address w/ 4K-entry selector), 6-cycle mis-prediction penalty
Hardware prefetch	Four stream prefetchers per core, 16 cache block prefetch distance, 2 prefetch degree; implementation follows [21]
On-chip network	4×4 and 8×8 2D mesh for 16- and 64-core CMP, respectively; runs at half the core's clock frequency; 1-cycle router latency, 1-cycle inter-router wire latency; XY-YX routing (O1TURN [22])
On-chip caches per core	32KB 4-way L1 I-/D- caches with a 1-cycle latency; 512KB 8-way unified L2 cache with a 4-cycle tag latency and a 12-cycle data latency; all caches use LRU replacement with the write-back policy and have a 64B block size
Cache coherence	Directory based MESI protocol, similar to SGI Origin 2000 [23] with on-chip directory cache and cache-to-cache transfer
On-chip directory cache	8K sets (for 16-core CMP) / 32K sets (for 64-core CMP) and 16-way dist. sparse directory [24] with a 5-cycle latency for private L2 cache models, LRU replacement; In-cache directory for the shared L2 cache model
DRAM	DDR3-1600 timing; t_{CL}=13.75ns, t_{RCD}=13.75ns, t_{RP}=13.75ns, $BL/2$=5ns; 8 banks, 2KB row-buffer per bank
L2 miss latency	Uncontended: {row-buffer hit: 25ns (100 cycles), closed: 42.5ns (170 cycles), conflict: 60ns (240 cycles)} + network latency
DRAM controller	Two/four independent controllers for 16-/64-core CMP, respectively; each controller has 12.8GB/s bandwidth and four ports; each port is connected to four adjacent cores (top four and bottom four/eight cores in 16-/64-core CMP)

Table 2. Baseline CMP configuration for 16 cores and 64 cores.

this case, all coherence processing is done by the directory. When the owner cache denies a broadcast lock request (because the data block has been migrated to the owner core by a previous broadcast), the directory waits for the request from the owner core to synchronize the coherence state between the directory and the owner cache. Note that the owner sends a coherence request (e.g., MESI protocol packets) to the directory as well as a broadcast request to neighbor cores to maintain coherence. Once the coherence request from the owner arrives at the directory, it processes the owner's request first, then the other requests.

3.4. Partitioning with Quality of Service (QoS)

Some threads may lose performance if they yield capacity (in their local L2 slice) to other threads. This subsection considers how to augment the partitioning algorithm to honor *quality of service* (QoS) for each thread. We define QoS as the maximum allowed performance degradation due to partitioning, similar to [16, 18]. The goal is to maximize overall performance and meet the QoS requirement for each thread.

In the following equations, "BC" stands for base execution cycle, "CC" is current cycle (i.e., the monitoring period), H_i is hit count in ith way, ML is L2 miss latency, F_s is the monitoring set ratio ($\#\ total\ sets/\#\ monitoring\ sets$), n is the number of cache ways allocated to a program in the current monitoring period, K is the associativity of one cache slice, and EC_j is expected cycles with cache capacity j. It is straightforward to modify our cache capacity allocation algorithm to provide this minimum cache capacity to each home core. Note that we apply C_{QoS} only to those cores with less than K total tokens ($j < K$).

$$BC = \begin{cases} CC + \sum_{i=K+1}^{n} H_i \times F_s \times ML & \text{if } n > K \\ CC & \text{if } n = K \\ CC - \sum_{i=n+1}^{K} H_i \times F_s \times ML & \text{if } n < K \end{cases} \quad (1)$$

$$EC_j = BC + \sum_{i=j+1}^{K} H_i \times F_s \times ML \quad (2)$$

$$C_{QoS} = MIN(j) \quad \text{where } EC_j \times (1 - QoS) < BC \quad (3)$$

Type	Benchmark
H	462.libquantum, 470.lbm, 459.GemsFDTD
MH	483.sphinx3, 429.mcf
M	433.milc, 437.leslie3d, 471.omnetpp, 403.gcc, 436.cactusADM
ML	454.calculix, 401.bzip2
L	473.astar,456.hmmer,435.gromacs,464.h264ref, 445.gobmk, 400.perlbench,416.gamess,450.soplex,444.namd,465.tonto

Table 3. Benchmark classification.

To determine if the QoS of a thread is satisfied, CloudCache needs to first estimate the thread's "base execution cycle". This time is the thread's execution time if it were given a single, private cache. Equation 1 estimates the base execution cycle. Equation 2 calculates the estimated execution cycles after allocating a certain cache capacity, j. The next step is to allocate minimum cache capacity to satisfy the QoS constraint based on the estimated baseline execution time as achieved by Equation 3.

4. Evaluation

4.1. Experimental setup

We evaluate CloudCache with a detailed trace-driven CMP architecture simulator [25]. The parameters of the machine we model are given in Table 2. We simulate a current generation 16-core CMP and a futuristic 64-core CMP. For cache coherence, we employ a distributed on-chip directory organization placed in all tiles. Directory accesses are hashed by the least significant address bits above the cache block offset.This fully distributes directory accesses. The number of directory entries is the same as the aggregated L2 cache blocks and the associativity of the directory is twice that of the L2 cache. This directory configuration is cost- and performance-effective for the workloads that we study.

Workloads. We characterized cache utilization of the SPEC CPU2006 benchmarks[2]; the results are summarized in Table 3. Based on misses per 1K instructions (MPKI), we classified the benchmarks into five types: Heavy (H), Medium-

[2]A few benchmarks are not included because we were unable to generate meaningful traces due to limitations in the experimental setup.

Workload	Composition	Benchmarks
Light1	all Ls	astar(2), hmmer(2), gromacs(2), h264ref(2), perlbench(2), gamess(2), soplex, namd, gobmk, tonto
Light2	ML + L	calculix(2), gcc(2), bzip2(2), astar(2), hmmer(2), gromacs(2), h264ref(2), perlbench(2)
Light3	M + L	milc(2), omnetpp(2), astar(2), hmmer(2), gromacs(2), h264ref(2), perlbench(2), namd, tonto
Medium1	M + ML	milc(3), leslie3d(3), omnetpp(2), gcc(2), cactusADM(2), calculix(2), bzip2(2)
Medium2	MH + M	sphinx3(2), mcf(2), milc(3), leslie3d(2), omnetpp(3), gcc(2), cactusADM(2)
Medium3	MH+M+ML	sphinx3(2), mcf(2), milc(2), leslie3d, omnetpp, gcc(2), cactusADM(2), calculix(2), bzip2(2)
Heavy1	H + MH	libquantum(3), lbm(3), GemsFDTD(3), sphinx3(3), mcf(4)
Heavy2	H+MH+M	libquantum(2), lbm(2), GemsFDTD(2), sphinx3(2), mcf(2), milc(2), leslie3d(2), omnetpp(2)
Heavy3	all Hs	libquantum(6), lbm(5), GemsFDTD(5)
Comb1	H + L	libquantum(2), lbm(2), GemsFDTD(2), astar(2), hmmer(2), h264ref(2), gamess, namd, gobmk,tonto
Comb2	MH + L	sphinx3(2), mcf(2), astar(2), hmmer(2), gromacs(2), h264ref, perlbench, gamess, soplex, gobmk, tonto
Comb3	MH + L	sphinx3, mcf, astar, hmmer, gromacs, h264ref, gamess(2), soplex(2), namd(2), gobmk(2), tonto(2)

Table 4. Multiprogrammed workloads (number in parentheses is the number of instances).

Heavy (MH), Medium (M), Medium-Light (ML), and Light (L). From this classification, we generated a range of workloads (combinations of 16 benchmarks), as summarized in Table 4. Light, Medium, and Heavy workloads represent the amount of cache pressure imposed by a group of benchmarks. The combination workloads (Comb1–3) are used to evaluate CloudCache's benefits for highly heterogeneous workloads. Table 5 summarizes the multithreaded workload based on PARSEC [26]. We focus on 16-thread parallel regions with the large input sets.

We randomly map programs in a given workload to cores to avoid being limited by a specific OS policy. All experiments use the same mapping. For the 16-core CMP configuration, one instance of each workload is evaluated. For the 64-core CMP configuration, we use multiple workload instances (1, 2, and 4) to mimic various processor utilization scenarios. We evaluate 25%, 50% and 100% utilization, where N% utilization means only N% of the total cores are active. We run each simulation for 1B cycles. We measure the performance with *weighted speedup* [27] to capture throughput against a private cache baseline. Weighted speedup is $\sum_i (IPC_i^{cache\_type} / IPC_i^{private\_cache})$.

Schemes for comparison. Our experiments compare performance of the five cache schemes: Shared cache, private cache, Dynamic Spill-Receive (DSR) [16], Elastic Cooperative Caching (ECC) [20] and CloudCache.[3] For intuitive presentation, results are given relative to a private cache. The shared cache has a distributed in-cache directory that maintains coherence between the shared L2 cache and the tile L1 caches. The other schemes are based on a private cache organization; thus, an on-chip distributed directory is used for coherence between main memory and the L2 caches (L1 caches are locally inclusive).

DSR was designed for a small-scale CMP with 4 to 16 cores [9]. A crossbar and a broadcast-based coherence protocol were used in the original proposal. We extended their work to a many-core CMP to objectively evaluate the benefit of DSR versus CloudCache. Similar to other private techniques in our evaluation, DSR is assumed to use the on-chip directory. DSR needs to transfer miss information to a spiller/receiver set's home tile whenever a miss occurs in the spiller/receiver set. Although this may incur network overhead, we do not model it. For the 64-core CMP, we reduced the number of spiller/receiver sets so that there are no overlapped monitoring sets.

[3] We also evaluated another recent proposal R-NUCA [10] but do not present its result for brevity. R-NUCA performance was similar to that of private cache for multiprogrammed workloads and that of shared cache for multithreaded workloads.

Workload	Benchmarks
Comb1	Blackscholes(*),Bodytrack(14),Facesim(*),Ferret(15)
Comb2	Blackscholes(*),Bodytrack(14),Canneal(15),Swaption(*)
Comb3	Blackscholes(*),Canneal(15),Facesim(*),Swaption(*)
Comb4	Bodytrack(14),Facesim(*),Ferret(15),Swaption(*)
Comb5	Canneal(15),Facesim(*),Ferret(15),Swaption(*)

Table 5. Multithreaded workloads evaluated (number in parentheses is the number of threads in the parallel region, '*'=16).

4.2. Results and Analysis

4.2.1. 16-core CMP

Figure 3 shows the results for the evaluation of CloudCache with the 16-core configuration. Figure 3(a) shows the average speedup of the shared cache, DSR, ECC, and CloudCache normalized to the baseline. CloudCache consistently outperforms the other techniques. The average speedup over the baseline is 1% (Heavy3) to 11% (Medium1). Some programs have a benefit at the expense of others. We call a program that gets more cache capacity than a single cache slice a "beneficiary program." A program that is given capacity smaller than a cache slice is a "benefactor."

Figure 3(b) and (c) illustrate the speedup of the beneficiaries and the slowdown of the benefactors. The error bars in these figures give the maximum value. The number above the bars in Figure 3(b) is the number of beneficiary benchmarks. For example, in Light2, there were one, five, three, and six benchmarks that experienced a speedup of 4%, 15%, 12%, and 23%, for the shared cache, DSR, ECC, and CloudCache, respectively. While CloudCache has better performance in terms of average speedup, the performance improvement for the beneficiary benchmarks is much higher

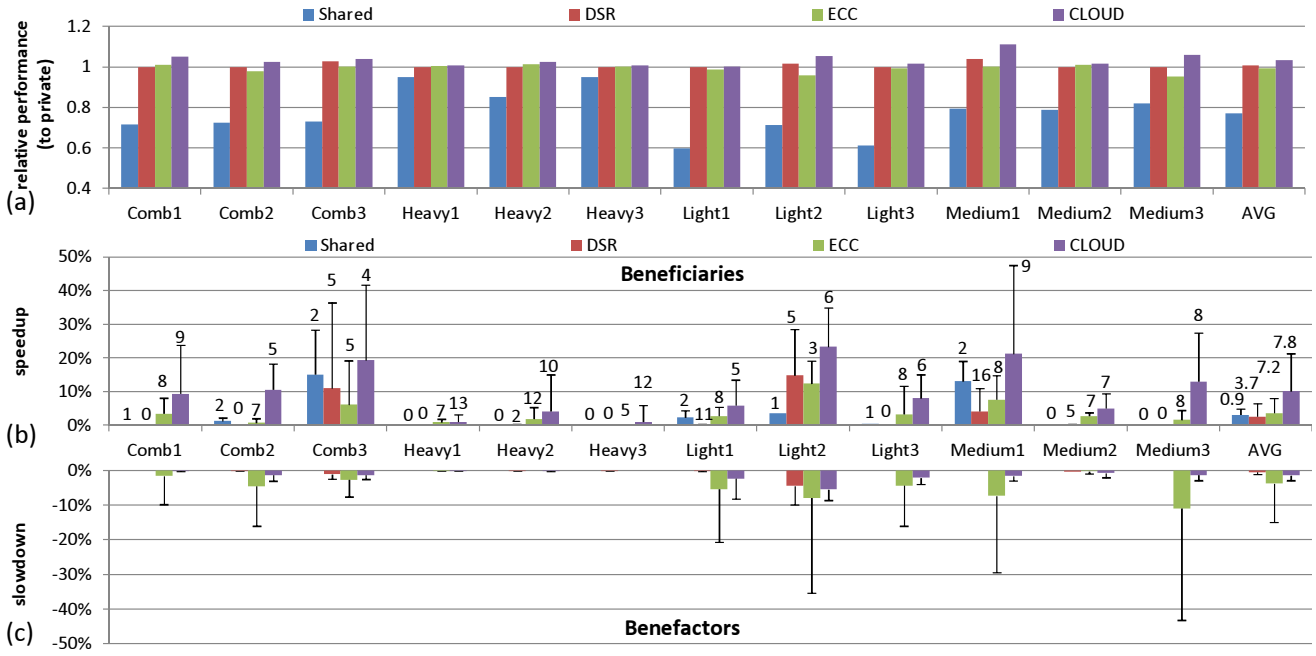

Figure 3. Performance of 16-core CMP. (a) Relative performance to the baseline (private cache). (b) Speedup of beneficiaries. (c) Slowdown of benefactors. Error bars show the maximum speedup of benchmarks in each workload for (b) and (c). Numbers above bars in (b) are the number of beneficiaries in each workload with the given techniques.

than the other techniques.

Furthermore, CloudCache does not significantly hurt the benefactors to improve performance of the beneficiary programs. CloudCache's average and worst case slowdown is limited to 5% and 9%, respectively (Light2). DSR and ECC have 9% and 45% slowdown in the worst case. Calculix in Light2, Medium1, and Medium3 performed worse with ECC, whose slowdown in each workload was 35%, 30%, and 43%. We found that calculix uses only 4 to 5 ways out of 8 ways. Because ECC does not allow programs with less than 6 ways to spill their evicted data [20], calculix's capacity was reduced too much. ECC's private cache capacity for each benchmark is determined by the hit counts in the LRU blocks of the private and shared areas. If the private area's LRU hit count for a given time (100K cycles as in [20]) is bigger than the shared area's LRU hit count, the private area is enlarged. However, benchmarks like calculix have a high hit count only for cache capacities that are above a specific large threshold. Once the cache capacity is reduced below the threshold capacity, a large LRU hit count will not be detected. Therefore, such programs never have a chance to gain more capacity. This is the limitation of local partitioning which fails to provide QoS. Global partitioning in CloudCache avoids this situation and gets better performance.

Interestingly, the shared cache has poor performance for all workloads and the degradation is magnified in three Light workloads because these workloads do not need much capacity. Instead, they prefer fast cache access. For heavy workloads (Heavy1, 2, and 3), the shared cache achieves 85% to 95% of the private cache's performance. The private based techniques do not have much performance improvement over the shared cache for these workloads due to many off-chip references. These references require an expensive three step access (i.e., to the local L2 cache, the directory, and then the memory controller).

In summary, we conclude that CloudCache maximizes the performance of beneficiaries as well as the number of beneficiaries. At the same time, CloudCache minimizes the performance slowdown of benefactors.

4.2.2. 64-core CMP

25% utilization scenario. Figure 4 shows the performance of each technique with 25% utilization (i.e., 16 threads are run on a 64-core CMP). Three performance evaluations—relative performance, speedup of beneficiary benchmarks and slowdown of benefactor benchmarks—are illustrated in Figure 4(a), (b), and (c). CloudCache consistently outperforms the other techniques by 1% to 33%. For the beneficiary benchmarks, CloudCache achieves a 20% to 50% average speedup, except for Heavy3.

The larger capacity from the 64-core CMP gives more chance to improve the performance of each benchmark. The number of beneficiaries in each workload is much higher in the 64-core CMP case. While the number of beneficiaries is similar for DSR, ECC, and CloudCache as shown in Figure 4(b), CloudCache has a much higher average performance improvement for the beneficiaries than DSR

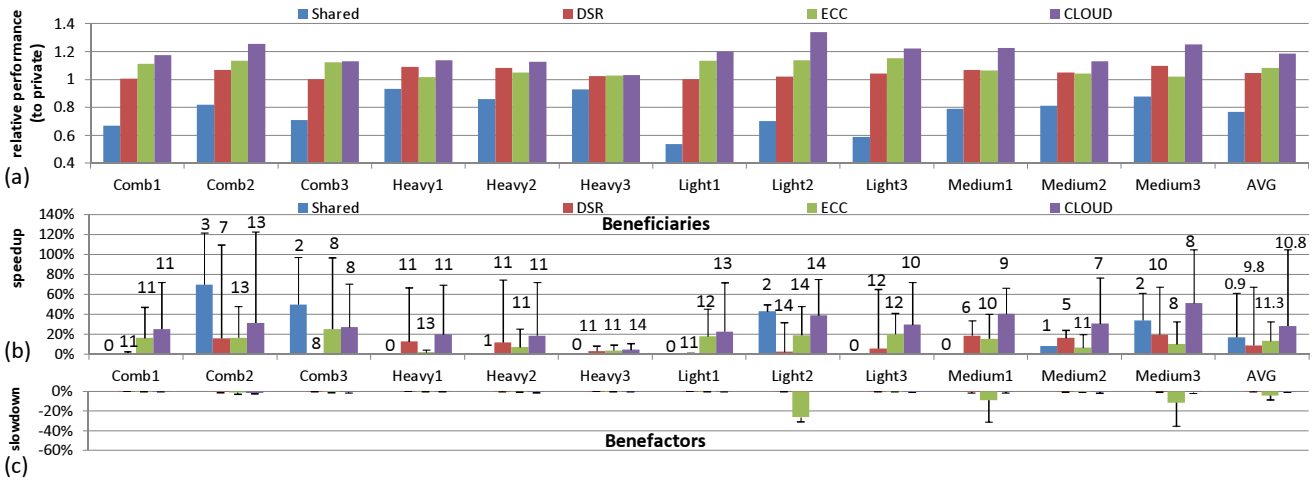

Figure 4. Performance of the 64-core CMP with 25% utilization. (a) Relative performance to the baseline (private cache) with 16 threads (25% utilization). (b) Speedup of beneficiaries. (c) Slowdown of benefactors. Meanings of error bars and numbers as before.

and ECC. Furthermore, all workloads, except Comb3 and Heavy2, have the best maximum performance improvement on the beneficiaries with CloudCache (error bars, Heavy2's maximum speedup of CloudCache is close to that of DSR). CloudCache offers a large capacity benefit as well as effective capacity isolation, such that it can provide optimized capacity to each benchmark.

Similar to the 16-core CMP experiments, ECC has severe problems with QoS. Figure 4(c) reveals this behavior. calculix in Light2, Medium1, and Medium3 has a large performance degradation of up to 35%. CloudCache limits the performance slowdown to only 2%.

The shared cache does better for some workloads (e.g., Comb2, Comb3, Light2 and Medium3) due to its large cache capacity. However, the number of beneficiaries is limited by capacity interference and a longer L2 access latency. Note that the performance of the shared cache is lower than the private cache.

50% and 100% utilization scenario. Figure 5 shows the performance of each technique with 50% and 100% utilization. While the average speedup is lower for 50% and 100% utilization than 25%, CloudCache clearly outperforms the other techniques. For Light, Medium, and Comb, CloudCache has 4% to 20% performance improvement over the private cache for 50% utilization (Figure 5(a)) and 4% to 17% improvement for 100% utilization (Figure 5(b)).

For the Heavy workloads, CloudCache has a 2% to 5% performance improvement over the private cache, except Heavy3 at 100% utilization. This benchmark has the best performance with the private cache due to two characteristics: a small gain in hit count from more capacity and many off-chip accesses. A small capacity benefit minimizes the potential improvement from partitioning in DSR, ECC, and CloudCache. Furthermore, DSR, ECC, and CloudCache generate more cache coherence traffic. This causes more network contention and overhead that harms performance when there are many off-chip accesses. Nevertheless, among DSR, ECC, and CloudCache, CloudCache has the best performance in this severe condition.

The speedup of beneficiaries (Figure 5(c) and (d)) reveals more about how these techniques perform. On average, CloudCache has a 21% and 14.7% performance improvement for beneficiaries while DSR and ECC have less than 10%. This result shows that CloudCache's global partitioning strategy gives more capacity to beneficiaries, which in turn boosts performance more than simple sharing (DSR) or local partitioning (ECC). Interestingly, the shared cache has a large improvement for Comb3's beneficiaries at 50% utilization, while there are no beneficiaries at 100% utilization. This result implies that simple capacity sharing is vulnerable to capacity interference in heavily loaded situations.

Multithreaded workloads. Figure 6 plots the performance of the multithreaded workloads on the 64-core CMP (four 16-threaded PARSEC benchmarks). The average speedup of the five workloads in Figure 6(a) shows that CloudCache does better than the other cache management techniques. The performance improvement over the private cache is 18% (Comb2) to 45% (Comb4).

Unlike multiprogrammed workloads, the shared cache does well for some cases (Comb1 and Comb4). It does not duplicate shared data blocks, and thus, the overall effective capacity is larger than the private cache. Figure 6(b) illustrates the speedup of individual PARSEC benchmarks for Comb2 and Comb5. In Comb2, blackscholes and canneal compete to get more capacity. DSR and ECC fail to improve canneal's performance. Shared and CloudCache get a benefit because they can better exploit the cache capacity. CloudCache's performance follows the shared cache for canneal in Comb2. Although DSR and ECC achieve speedup for blackscholes, CloudCache's performance improvement

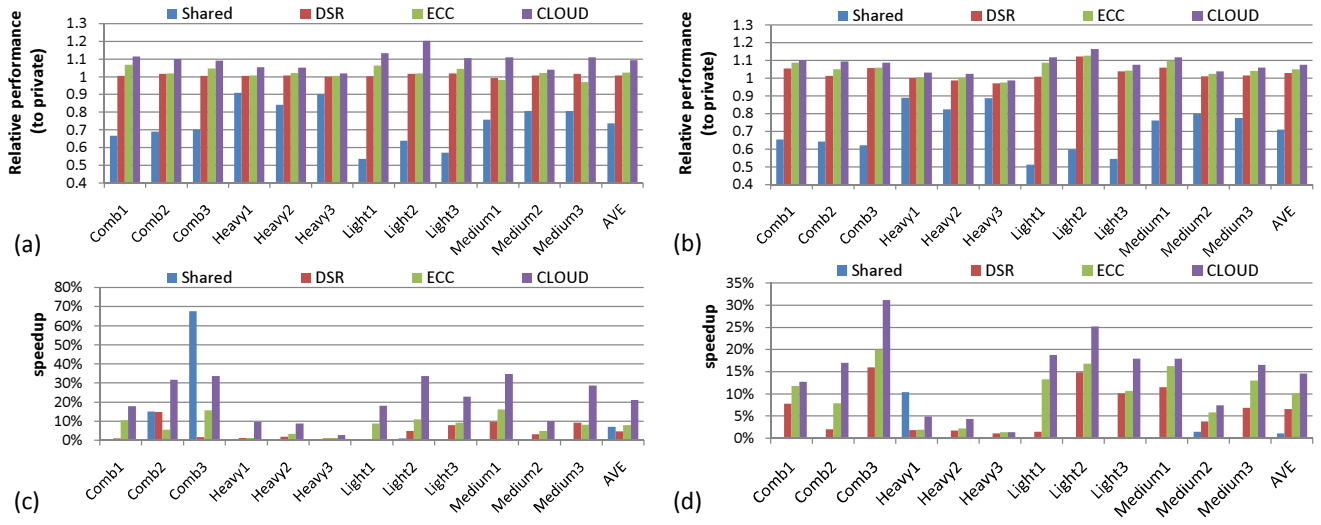

Figure 5. Performance of the 64-core CMP with 50% (32 threads) and 100% (64 threads) utilization. (a)–(b) Average speedup over the baseline (private cache). (c)–(d) Speedup of beneficiaries.

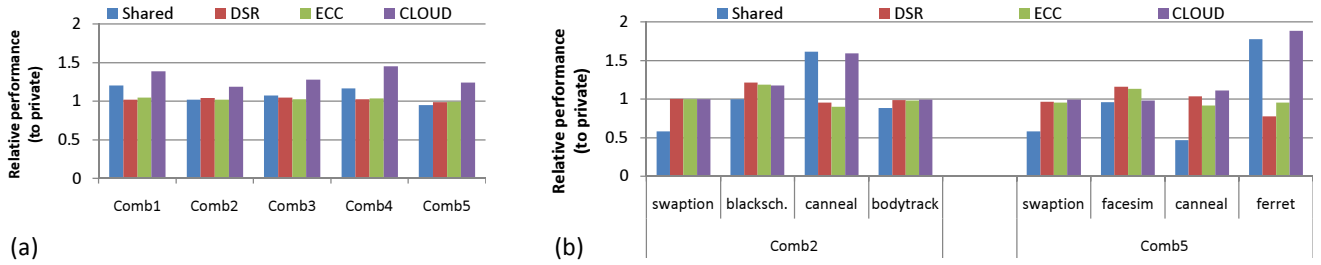

Figure 6. Performance of the 64-core CMP with multithreaded workloads (four PARSEC benchmarks, 16 threads each). (a) Average speedup over the baseline (private cache). (b) Speedup of each benchmark in Comb2 and Comb5.

is close to these techniques.

Comb5 has a different scenario. For facesim, CloudCache has a slight performance degradation while DSR and ECC have significant performance improvements. However, ferret and canneal have a greater benefit with CloudCache. Comparing Comb2 and Comb5 in Figure 6(b), CloudCache's characteristic is clear: it always maximizes the cache capacity for the most beneficiaries, and each benefactor's slowdown is minimized. Canneal has a 58% perfomance improvement in Comb2 and a 12% speedup in Comb5. CloudCache allocates much more capacity to ferret for Comb5, and thus, canneal cannot be improved as much as in Comb2.

We conclude that CloudCache's global partitioning is beneficial for a large aggregated cache capacity. Distance-aware placement and limited target broadcast also effectively cooperate to boost performance.

Quality of Service support. Let us examine the performance of multiprogrammed workloads on the 16-core CMP with the QoS support. Figure 7(a) presents the average speedup of CloudCache with three QoS levels, no QoS, 5% QoS, and 2% QoS. 5% (2%) QoS means the maximum allowed performance degradation is 5% (2%) of the private cache. The figure shows the QoS support does not significantly decrease overall performance.

Figure 7(b) and (c) are S-curves of each application's performance with the three QoS levels. As shown in Figure 7(b), the QoS support does not significantly decrease the performance of beneficiaries. Figure 7(c) plots only the performance of the benefactors. In Figure 7(c), 5% QoS level meets all applications' performance requirement. For 2% QoS level, two programs have 2.2% performance slowdown. In these cases, the miss rate computation is somewhat inaccurate due to sampling. While the error is negligible, a more conservative design (e.g., by using a larger average miss latency in Equation 3) might better guarantee the QoS level.

4.2.3. Impact of individual techniques

Impact of dynamic global partitioning. Figure 8 depicts MPKI for sphinx, hmmer, and gobmk from Comb2. The figure illustrates the 25% utilization case with a representative execution period. From Figure 8(a), CloudCache has a significant partitioning benefit over the other techniques for

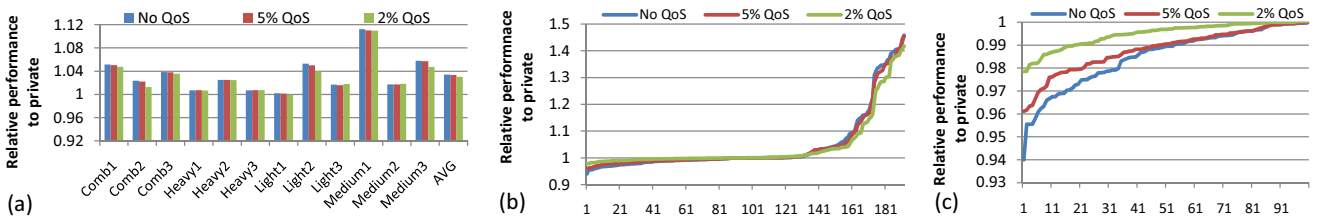

Figure 7. Performance with the QoS support. (a) Average speedup of CloudCache with three QoS levels (no QoS, 5% QoS, and 2% QoS) normalized to private cache. (b) S-curve of all programs. (c) S-curve of benefactors.

sphinx. The shared cache has a capacity benefit over Private and DSR. In this workload, DSR determines sphinx as a receiver, which does not spill data to other cores. With limited spilled data, a DSR receiver performs similar to a private cache. ECC has better performance because it can spill sphinx's evicted data to other shared cache regions. The limited data spilling from other cores may increase the benefit for sphinx. CloudCache's MPKI is significantly smaller than all other techniques because it dedicates a large cache capacity that is not subject to interference.

Figure 8(b) shows a case, hmmer, where CloudCache does not outperform the other techniques. The large MPKI for the shared cache shows that hmmer is greatly impacted from cache capacity interference while additional capacity might be helpful as shown in the ECC figure. CloudCache's MPKI is slightly higher than ECC for hmmer. This shows that the effective cache capacity of CloudCache is somewhat smaller than that of ECC. However, the difference between the two schemes for hmmer is limited.

Lastly, gobmk, shown in Figure 8(c), has the highest MPKI with CloudCache. CloudCache aggressively reduces the cache capacity of gobmk to help other benchmarks. However, the maximum MPKI of this benchmark is only 0.14, which is far smaller than that of sphinx (20) and hmmer (3.5). This illustrates that CloudCache's performance loss is limited for this benchmark. In fact, with distance-aware placement and limited broadcast, CloudCache even outperforms the other techniques for gobmk.

From this analysis, the benefit of CloudCache's global partitioning is apparent. First, it judiciously grants more cache capacity to benchmarks with more potential for performance improvement. The simple capacity sharing schemes (DSR and ECC) can generate capacity interference in the shared cache capacity, which in turn reduces the benefit of more capacity. Second, the effective use of cache capacity with CloudCache in moderate beneficiaries (e.g., hmmer) is close to the best technique (ECC). This leads to similar performance improvement with ECC and DSR. Third, CloudCache aggressively grants cache capacity from less sensitive benchmarks (e.g., gobmk) to more capacity sensitive ones. This achieves a better overall speedup without harming other benchmarks.

Impact of distance-aware data placement. We investigate the performance of the cache management techniques when only one benchmark is run on a 64-core CMP. This highlights the impact of additional capacity and distance-aware placement. We disabled CloudCache's broadcast function to show the pure effect of distance-aware placement. We make a few interesting observations. First, the shared cache performs the best for gcc, but it does the worst for many benchmarks. For example, GemsFDTD might need the additional capacity benefit from shared cache. However, shared cache's capacity benefit is offset by a longer NUCA latency. Gcc has many hits beyond the local L2 cache slice (i.e., 512KB), and thus, it is capacity demanding. In this situation, the shared cache can directly determine the data location and does not need three-step communication involving the directory, unlike DSR, ECC, and CloudCache.

Sphinx3 is also an interesting example: the shared cache does better than DSR and ECC. However, it does worse than CloudCache. Like gcc, sphinx is capacity demanding but it has a sharp fall-off in the hit counts once a particular capacity is reached. As a result, distance-aware placement is beneficial because it can concentrate hits in nearby cache slices. For gcc, it is more important to add additional capacity than to keep the hits near the home core.

The other benchmarks, except milc, have the best performance with CloudCache. While milc does not achieve performance improvement with all the techniques, CloudCache's additional network traffic causes a small performance slowdown. Note this is the case when milc is executed alone in a 64-core CMP. In real conditions, when milc is run with other benchmarks, milc has limited cache capacity which naturally does not generate additional network traffic. Interestingly, ECC performs worse than DSR for most benchmarks.

Impact of limited target broadcast. We also investigate the performance improvement from the limited target broadcast technique. This experiment is performed with one thread in a 64-core CMP so that the full performance impact of broadcast can be revealed. This experiment uses varying broadcast depth from 1 to 5 hops, which is the maximum distance of cores that are targets of a broadcast to the home core.

The benchmarks are roughly clustered in three categories. First, benchmarks such as bzip2, gromacs, calculix, hmmer, h264ref, omnetpp, astar, and sphinx3 benefit significantly

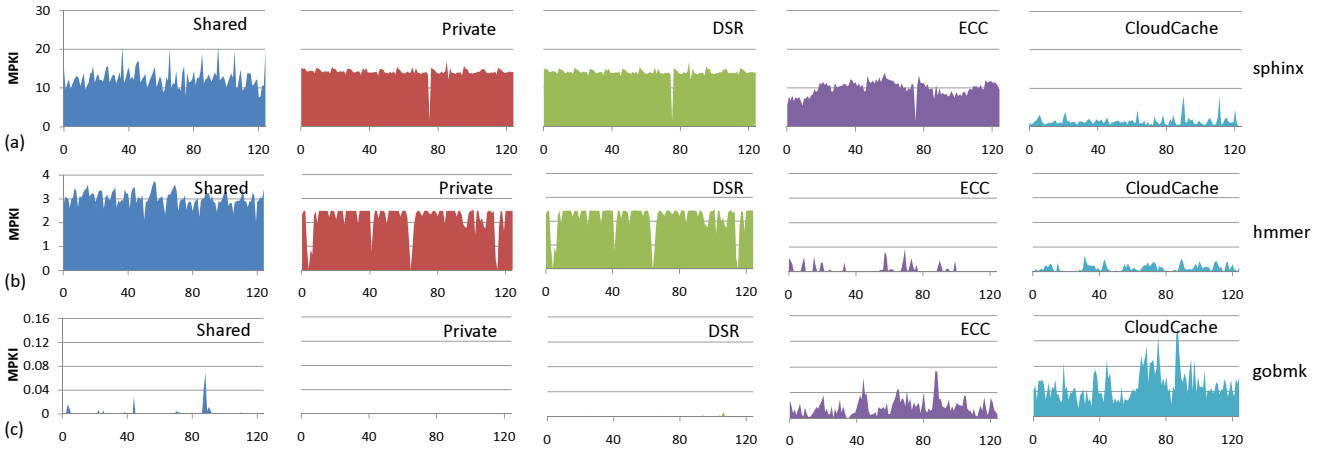

Figure 8. MPKI of cache management techniques for three benchmarks (sphinx, hmmer, and gobmk) in Comb2 with 25% utilzation of the 64-core CMP. X-axis unit is million instructions. (a) Sphinx: CloudCache < ECC < Shared < DSR = Private. (b) Hmmer: ECC < CloudCache < DSR = Private < Shared. (c) Gobmk: Private < DSR < Shared < ECC < CloudCache.

from limited broadcast. They have up to 16% performance improvement. The peak performance benefit was usually achieved with a broadcast depth of two. A deeper depth incurs more traffic, which reduces the benefit of broadcast.

Second, there are benchmarks, like gcc, mcf, milc, leslie3d, libquantum, and lbm, whose performance is hurt by broadcast. The performance loss is even more apparent at a 3-hop depth. With a large depth, performance slowdown of up to 11.5% was observed (milc). This illustrates that broadcast is not always good due to its additional network traffic. However, under realistic workloads, CloudCache would actively adjust the capacity of these benchmarks to be small, which would automatically limit this effect.

Lastly, benchmarks such as perlbench, gamess, namd, gobmk, and soplex are relatively insensitive to broadcast. These benchmarks have a small number of remote cache accesses, and thus, the impact of the broadcast is limited.

4.2.4. Putting all techniques together

Figure 9 presents the L2 access latency profile of bzip2 ("ML" type, see Table 3) and sphinx3 ("MH" type). Comparing the shared and private cache, we observe the trade-off between on-chip cache miss rate (shared cache is better) and on-chip cache access latency (private cache has many local hits). DSR, ECC, and CloudCache (without limited target broadcast) share the strength of a private cache and have many local cache hits. Furthermore, many accesses are satisfied from remote cache capacity. Note that the accesses in CloudCache have lower latency due to distance-aware placement. With limited target broadcast, the non-local cache hit latency is even smaller.

The performance gap between the shared cache and the other schemes is smaller with sphinx, which requires much more capacity for high performance than bzip2 (i.e., data reuse distance is longer). Therefore, the private cache suffers from a high cache miss rate. Many cache accesses are serviced by remote cache slices in DSR, ECC, and CloudCache. While CloudCache's distance-aware placement helps, its benefit is somewhat limited as many cache slices are involved. Nevertheless, limited target broadcast significantly improves performance by 8%.

5. Conclusion

Future CMPs are expected to have many cores and cache resources. We showed in this work that both efficient capacity partitioning and effective NUCA latency mitigation are required for scalable high-performance on a many-core CMP. We proposed CloudCache, a novel scalable cache management substrate that achieves three main goals: minimizing off-chip accesses, minimizing remote cache accesses, and hiding the effect of remote directory accesses. CloudCache encompasses dynamic global partitioning, distance-aware data placement, and limited target broadcast. We extensively evaluate CloudCache's performance with two basic techniques (shared and private caches) and two recent proposals (DSR and ECC). CloudCache outperforms the other techniques by up to 18% in comparison to the best one. Our detailed analysis demonstrates that our proposed techniques significantly improve system performance. We also showed that CloudCache naturally accommodates QoS support.

References

[1] M. Azimi et al. Integration challenges and trade-offs for tera-scale architectures. *Intel Tech. J.*, 11(3):173–184, August 2007.

[2] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *Intel Tech. J.*, 27(3):1–15, August 2008.

[3] Tilera. Tilera announces the world's first 100-core processor with the new tile-gx family. `http://www.tilera.com/news_&_events/press_release_091026.php`.

[4] Tensilica. Tensilica - servers, storage, and communications

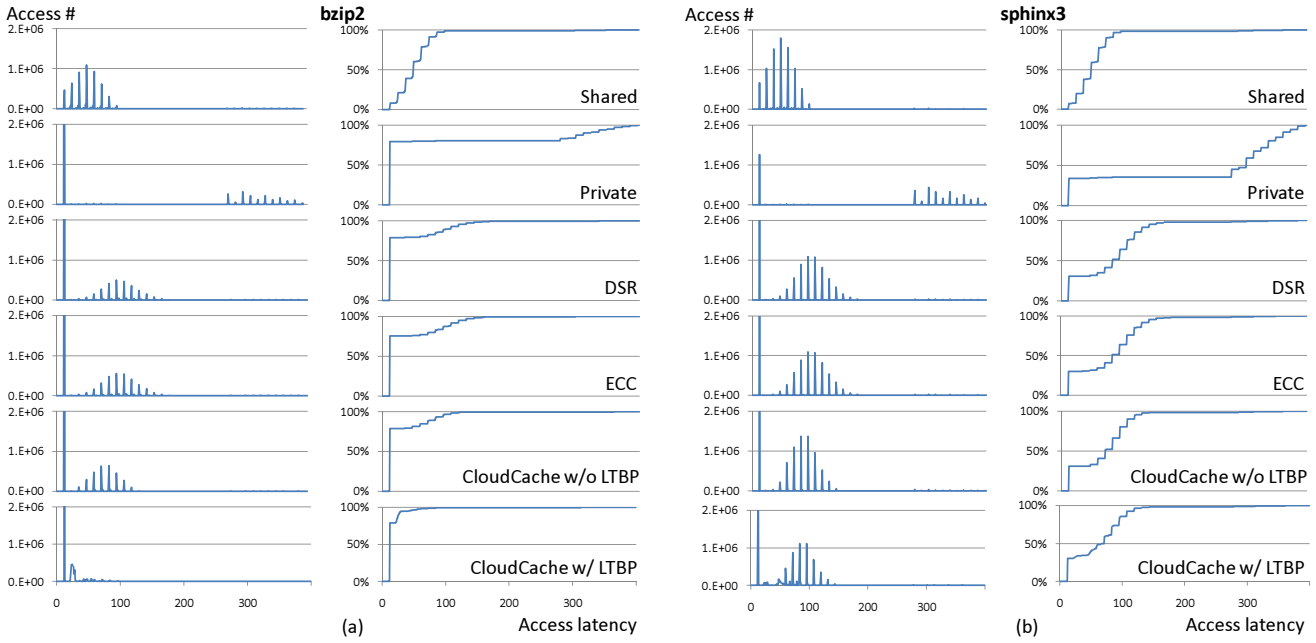

Figure 9. Access latency distribution (left) and cumulative distribution (right). (a) bzip2. (b) sphinx3.

infrastructure. `http://www.tensilica.com/markets/networking-storage.htm`.

[5] M. Zhang et al. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. *ISCA*, 2005.

[6] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in cmps. *ISCA*, 2005.

[7] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. *ISCA*, 2006.

[8] B Beckmann et al. Asr: Adaptive selective replication for cmp caches. *MICRO*, 2006.

[9] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in cmps. *HPCA*, 2009.

[10] H. Hardavellas et al. Reactive nuca: near-optimal block placement and replication in distributed caches. *ISCA*, 2009.

[11] D. Kaseridis et al. Bank-aware dynamic cache partitioning for multicore architectures. *ICPP*, 2009.

[12] A. Esser. Best practices for unlocking your hidden data center. `http://www.dell.com/downloads/global/power/ps1q08-20080198-Esse.pdf`.

[13] J. M. Kaplan et al. Revolutionizing data center energy efficiency. `http://www.mckinsey.com/clientservice/btopointofview/pdf/Revolutionizing_Data_Center_Efficiency.pdf`.

[14] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches,. *MICRO*, 2004.

[15] J. Huh et al. A nuca substrate for flexible cmp cache sharing. *ICS*, 2005.

[16] M. K. Qureshi et al. Utility-based partitioning of shared caches. *MICRO*, 2006.

[17] M. R. Marty et al. Virtual hierarchies to support server consolidation. *ISCA*, 2007.

[18] K. J. Nesbit et al. Virtual private caches. *ISCA*, 2007.

[19] H. Lee et al. Stimuluscache: Boosting performance of chip multiprocessors with excess cache. *HPCA*, 2010.

[20] E. Herrero et al. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. *ISCA*, 2010.

[21] J. Tendler et al. Power4 system microarchitecture. *IBM Technical White Paper*, October 2001.

[22] D. Seo et al. Near-optimal worst-case throughput routing for two-dimensional mesh networks. *ISCA*, 2005.

[23] M. Plakal et al. Lamport clocks: Verifying a directory cache-coherence protocol. *SPAA*, 1998.

[24] A. Gupta et al. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. *ICPP*, 1990.

[25] H. Lee et al. Two-phase trace-driven simulation (tpts): A fast multicore processor architecture simulation approach. *Software: Practice and Experience (SPE)*, March 2010.

[26] C. Bienia et al. The parsec benchmark suite: Characterization and arch. implications. *PACT*, 2008.

[27] A. Snavely et al. Symbiotic job scheduling for a simultaneous multithreading processor. *ASPLOS*, 2005.