Sharing-aware Efficient Private Caching in Many-core Server Processors

Sudhanshu Shukla Mainak Chaudhuri
Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, India
{sudhan, mainakc}@cse.iitk.ac.in

Abstract— **The general-purpose cache-coherent many-core server processors are usually designed with a per-core private cache hierarchy and a large shared multi-banked last-level cache (LLC). The round-trip latency and the volume of traffic through the on-die interconnect between the per-core private cache hierarchy and the shared LLC banks can be significantly large. As a result, optimized private caching is important in such architectures. Traditionally, the private cache hierarchy in these processors treats the private and the shared blocks equally. We, however, observe that elimination of all non-compulsory non-coherence core cache misses to a small subset of shared code and data blocks can save a large fraction of the core requests to the LLC indicating large potential for reducing the interconnect traffic in such architectures.**

We architect a specialized exclusive per-core private L2 cache which serves as a victim cache for the per-core private L1 cache. The proposed victim cache selectively captures a subset of the L1 cache victims. Our best selective victim caching proposal is driven by an online partitioning of the L1 cache victims based on two distinct features, namely, an estimate of sharing degree and an indirect simple estimate of reuse distance. Our proposal learns the collective reuse probability of the blocks in each partition on-the-fly and decides the victim caching candidates based on these probability estimates. Detailed simulation results on a 128-core system running a selected set of multi-threaded commercial and scientific computing applications show that our best victim cache design proposal at 64 KB capacity, on average, saves 44.1% core cache miss requests sent to the LLC and 10.6% execution cycles compared to a baseline system that has no private L2 cache. In contrast, a traditional 128 KB non-inclusive LRU L2 cache saves 42.2% core cache misses sent to the LLC compared to the same baseline while performing slightly worse than the proposed 64 KB victim cache. In summary, our proposal outperforms the traditional design and enjoys lower interconnect traffic while halving the space investment for the per-core private L2 cache. Further, the savings in core cache misses achieved due to introduction of the proposed victim cache are observed to be only 8% less than an optimal victim cache design at 32 KB and 64 KB capacity points.

Index Terms— **Many-core server processors; Private victim caches; Sharing-aware private caching.**

I. Introduction

The emerging many-core server processors with tens of cores are equipped with a per-core private cache hierarchy and a large multi-banked on-die shared last-level cache (LLC). The cores along with their private cache hierarchy and the LLC banks are distributed over a scalable on-die interconnect. Due to the traversal through the interconnect, the core cache miss requests can experience large average round-trip latencies even if they hit in the LLC. As the system grows in terms of core-count, the average round-trip LLC hit latency as well as the volume of traffic in the interconnect typically increase making efficient private caching an important requirement for such systems. In this paper, we squarely focus on the problem of architecting an efficient private cache hierarchy for many-core server processors running multi-threaded workloads drawn from the domains of commercial computing (web serving and data serving) and scientific computing. Traditionally, a two-level private cache hierarchy is used per core where the private L1 and L2 caches treat the private and shared blocks equally. We start our exploration with a baseline design that does not have a private L2 cache allowing us to understand the properties of the L1 cache misses. Our approach is to characterize the cores' L1 cache misses that hit in the LLC and exploit this run-time characterization to eliminate a subset of these misses by architecting a specialized space-efficient private L2 cache.

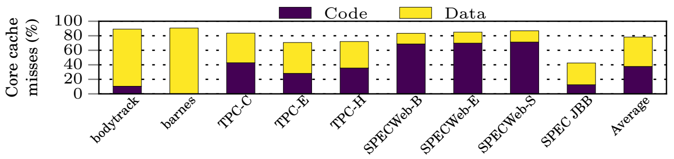

Fig. 1. Non-compulsory non-coherence core cache misses that hit in LLC.

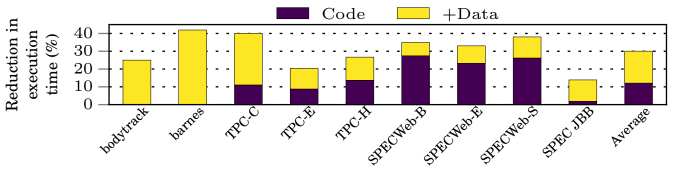

Fig. 2. Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

To quantify the potential performance improvement achievable by optimizing the core caches, we conduct an experiment on a simulated 128-core server processor with each core having private instruction and data L1 caches (32 KB 8-way each) and a shared 32 MB 16-way LLC partitioned into 128 set-interleaved banks. A 256 KB 16-way LLC bank is attached to a core slice and the 128 slices are arranged in a 16 × 8 mesh interconnect exercising dimension-order-routing and having a four-stage routing pipeline at each switch clocked at 2 GHz.[1] In this experiment, the non-compulsory non-coherence L1 cache misses which hit in the LLC are assumed to hit in the L1 cache i.e., they are charged only the L1 cache lookup latency and do not generate any interconnect traffic. We note that the compulsory and coherence misses cannot be reduced in number by optimizing the private cache hierarchy (assuming a fixed block size). Figure 1 shows the percentage core cache misses saved in this experiment partitioned into code and data misses. On average, 78% core cache misses can be saved. For the web and data serving workloads (TPC, SPEC Web, and SPEC JBB), both code and data contribute significantly to the saved misses, while for the remaining applications, the savings primarily arise from data accesses. Figure 2 shows the percentage reduction in execution time when this optimization is applied to only code misses and additional reduction when it is applied to both code and data misses. On average,

[1] Further details of our simulation environment can be found in Section IV.

30% execution time can be optimized away by eliminating the non-compulsory non-coherence code and data L1 cache misses that hit in the LLC. The savings in the execution time range from 14% (SPEC JBB) to 40% (barnes and TPC-C). When only the non-compulsory non-coherence code misses which hit in the LLC are eliminated, the average saving in execution time is 12%. The savings in execution time correlate well with the volume of saved misses shown in Figure 1.

Motivated by this large potential improvement in performance, we thoroughly characterize the core cache misses that hit in the LLC (Section II). Our characterization study reveals that a small subset of shared code and data blocks contributes to a large fraction of the core cache misses that hit in the LLC. This observation leads us to explore the design of a per-core private L2 cache that can serve as an efficient victim cache for the private L1 cache (Section III). The goal of our victim cache designs is to capture the critical subset of the shared blocks. Our best proposal classifies the L1 cache victims into distinct partitions based on two features, namely, an estimate of sharing degree and a simple indirect measure of reuse distance. The collective reuse probability of each partition is learned on-the-fly and used to decide if the L1 cache victims belonging to a partition should be inserted in the victim cache. To the best of our knowledge, this is the first sharing-aware private cache hierarchy design proposal for many-core server processors. Simulation results obtained from a detailed model of a 128-core server processor (Section IV) show that our best victim cache design with 64 KB capacity saves 44.1% core cache misses sent to the LLC and 10.6% execution cycles, on average slightly outperforming a traditional non-inclusive per-core 128 KB L2 cache exercising LRU replacement policy (Section V). Further, the savings in core cache misses achieved by our best victim cache proposal are observed to be only 8% less than an optimal victim cache design at 32 KB and 64 KB capacity points.

II. Characterization of Core Cache Misses

In this section, we analyze the non-compulsory non-coherence L1 cache misses that hit in the LLC. Since both code and data have important contributions to these misses, this analysis must characterize these misses using features other than code and data. We begin by partitioning these misses based on the sharing types of the LLC blocks being accessed. The LLC block types are discussed below. An LLC block is said to be temporally private if it never experiences any kind of sharing between more than one core at the same time. A core X accesses such a block from the LLC and caches it privately. It is evicted from the private cache hierarchy of core X before the next LLC access (from the same core X or from a different core Y) to the block. All other LLC blocks are said to be shared. We partition the shared blocks into two groups based on the degree of sharing. We attach a **S**hared **R**ead **A**ccess (SRA) counter with each block to measure its degree of sharing. The SRA counter of a block is initialized to zero when it is filled into the LLC from the main memory. This counter is incremented for a block when an LLC read access (due to a core cache data load or code read miss) to the block hits in the LLC and finds the block in the shared state (S state in MESI coherence protocol). All temporally private blocks have zero SRA. We put all shared blocks with SRA=0 in one group (low degree of sharing) and the remaining shared blocks in another group.

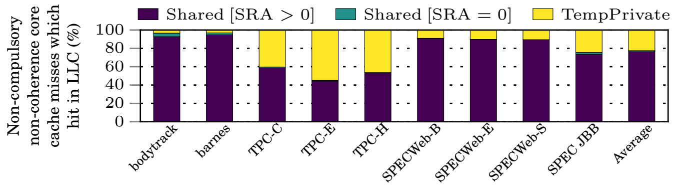

Fig. 3. Distribution of non-compulsory non-coherence core cache misses that hit in LLC based on the sharing types of the LLC blocks being accessed.

Figure 3 shows the distribution of non-compulsory non-coherence L1 cache misses that hit in the LLC. "TempPrivate" represents the temporally private category. On average, 76% of these misses access shared blocks with positive SRA. For all the applications, except TPC-E, more than half of these misses fall in this category. Figure 4 shows the percentage reduction in execution time when these misses are treated as L1 cache hits. The bottom segment of each bar shows the percentage reduction in execution time when only the core cache misses to the shared LLC blocks with positive SRA are treated as L1 cache hits. The middle segment of each bar shows the additional saving in execution time when the core cache misses to the shared LLC blocks with zero SRA are also treated as L1 cache hits. The top segment of each bar shows the additional saving in execution time when the core cache misses to the temporally private LLC blocks are also treated as L1 cache hits. On average, 19% execution time can be optimized away by saving the core cache misses which hit the shared LLC blocks with positive SRA. Saving the core cache misses to the shared LLC blocks with zero SRA has negligible impact on performance. These results clearly highlight that saving the core cache misses to the shared LLC blocks with positive SRA is important for performance and interconnect traffic. Figure 5 quantifies the percentage of the LLC blocks that are shared and have positive SRAs. On average, just 12% of the LLC blocks fall in this category. Barnes is a clear outlier with 78% of the LLC blocks in this category. Among the rest, only bodytrack and TPC-H have more than 2% of the LLC blocks that are in this category. Therefore, on average, only 12% of the LLC blocks contribute to 76% of the core cache misses that hit in the LLC.

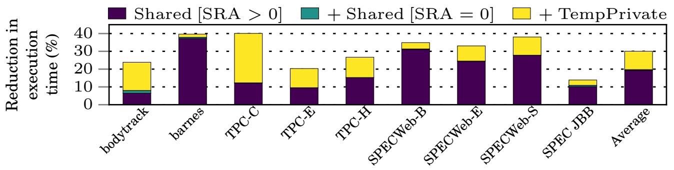

Fig. 4. Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

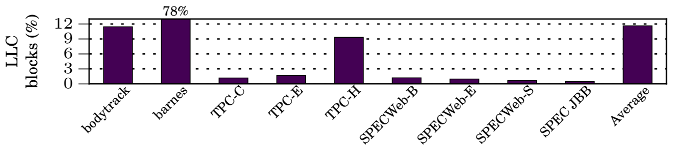

Fig. 5. Percentage of allocated LLC blocks that are shared with positive SRA.

We further classify the shared LLC blocks with positive SRA based on a normalized SRA ratio. The SRA ratio for an LLC block at any point in time is defined as the ratio of the SRA counter value to the total number of LLC accesses to the block arising from the core cache misses. We classify the shared LLC blocks with positive SRA into three SRA ratio categories, namely, C_1, C_2, and C_3. The C_1 category includes all LLC blocks with SRA ratio $\in (0, \frac{1}{2}]$. For C_2 and C_3, the SRA ratio ranges are $(\frac{1}{2}, \frac{3}{4}]$, and $(\frac{3}{4}, 1]$, respectively. Figure 6 shows the distribution of the shared LLC blocks with positive SRA. Recall that only 12% of the LLC blocks are shared with positive SRA. Among these, on average, 49%, 10%, and 41% are in C_1, C_2, and C_3, respectively. Figure 7 shows the distribution of the non-compulsory non-coherence L1 cache misses that hit in the LLC. On average, 68% of these core cache misses access LLC blocks in C_3 category. This is an important piece of data showing that only 41% of 12% (or, overall 5%) LLC blocks cover 68% of non-compulsory non-coherence L1 cache misses that hit in the LLC. Therefore, it may be possible to capture a significant subset of these L1 cache misses by incorporating a specialized per-core victim cache. Figure 8 further shows the percentage reduction in execution time when non-compulsory non-coherence L1 cache misses that hit in the LLC are saved and treated as L1 cache hits. For each application, we show the gradual reduction in execution time as core cache misses to C_3, $C_3 + C_2 + C_1$, $C_3 + C_2 + C_1 +$ shared with zero SRA, and all LLC blocks are saved. On average, 15.5% execution time can be optimized away by saving the core cache misses to the C_3 blocks.

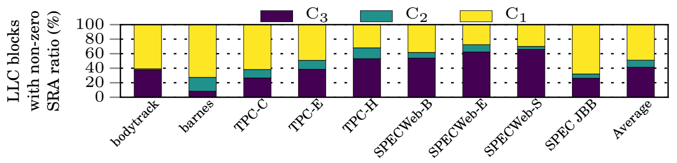

Fig. 6. Distribution of the shared LLC blocks into the SRA ratio categories.

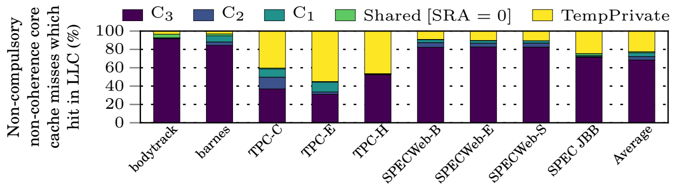

Fig. 7. Distribution of non-compulsory non-coherence core cache misses that hit in LLC based on the sharing status of the LLC block being accessed.

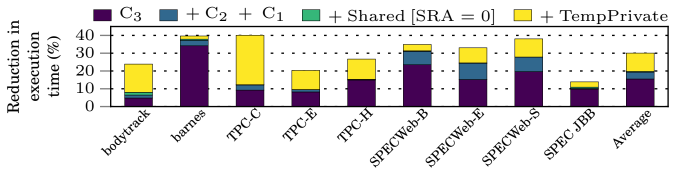

Fig. 8. Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

III. Victim Cache Design

In this section, we architect a private per-core unified victim cache (VC) to capture a subset of the L1 instruction and data cache victims. We begin our discussion by reviewing the basic VC architecture that admits all L1 cache victims (Section III-A). Next, we present two design proposals for selective victim caching (Section III-B) that exploit the findings of our characterization results. All the VC designs considered in this paper are 8-way set-associative. The L1 cache and the VC are looked up serially to avoid lengthening the L1 cache access latency. On an L1 cache miss, the VC is looked up. On a VC hit, the block is invalidated from the VC and copied to the L1 instruction or data cache depending on the request type. On a VC miss, the block is fetched from the outer levels of the memory hierarchy (LLC or main memory) and inserted into the L1 instruction or data cache. As a result, the VC is equivalent to a private per-core L2 cache that is exclusive of the L1 caches.

A. Victim Caching without Selection

The traditional VC architecture admits all L1 cache victims. We evaluate two replacement policies for such a VC. The first one evicts the least-recently-filled (LRF) block in a VC set.[2] This design requires three replacement state bits per block in an 8-way cache. This design will be referred to as LRF-VC. The second design devotes only one replacement state bit per block. This bit is set to one when a block is inserted into the VC. If all blocks in a set have this bit set to one, all the bits in that set are reset except the bit corresponding to the most recently filled block. Within a set, the replacement policy victimizes the block with the replacement state bit reset; a tie among multiple such eligible candidates is broken by victimizing the block with the lowest physical way id. This replacement policy will be referred to as not-recently-filled (NRF) and this design will be referred to as NRF-VC. The NRF policy is motivated by the observation that the first order locality of a block inserted in the VC is already filtered by the L1 cache and therefore, a precise fill order as maintained by the LRF policy may not be necessary to achieve good performance.

[2] A least-recently-used replacement policy has no meaning in a VC because on a VC hit, a block is invalidated.

B. Selective Victim Caching

The two selective victim caching proposals discussed below constitute the crux of our contributions. Our L1 cache miss characterization study has established that the selective victim caching proposals must primarily target the shared LLC blocks with positive SRA ratio and that the C_3 blocks are particularly important. In addition to the three categories (C_1, C_2, and C_3) of non-zero SRA ratio, we use C_0 to denote the category of shared as well as temporally private blocks with zero SRA ratio. To identify the category of an LLC block, the SRA ratio needs to be estimated online. For this purpose, two six-bit saturating counters, namely SRA Counter (SRAC) and **O**ther **A**ccess **C**ounter (OAC), are maintained for the block. The SRAC is incremented on LLC read accesses which find the block being requested in the shared state. The OAC is incremented on all other LLC accesses (except writeback) to the block. Both the counters of the block are halved when any of the counters has saturated. The SRA ratio estimate for the block is given by the fraction $\frac{SRAC}{SRAC+OAC}$. The sparse directory entry that tracks coherence of a block is extended by twelve bits to accommodate the two counters. Once a block returns to the unowned/non-shared state, the counters are reset and the SRA ratio for the block is deemed zero. Also, when the sparse directory entry of a block is evicted, its SRA ratio is assumed to become zero. When a block is fetched from the LLC into the L1 cache, the block's SRA ratio category is also fetched and maintained by extending the L1 cache tag by two bits. A block fetched from main memory (due to LLC miss) is assumed to belong to category C_0. When an L1 cache block is evicted, its SRA ratio category is used in deciding whether the block should be admitted into the VC, as discussed in the following designs. If an L1 cache victim is allocated in the VC, its SRA ratio category is also maintained in the VC by extending the VC tag by two bits. When a VC entry is copied into the L1 cache, its SRA ratio category is also copied.

1) SRA-gNRF-VC: Our first selective VC design tries to capture a subset of the high SRA ratio blocks and implements a generational NRF (gNRF) replacement policy. First, we discuss the gNRF policy. The design divides the entire execution into intervals or generations. Each VC entry is extended with two state bits, namely, a fill (F) bit and an eviction priority (EP) bit. When a VC block is filled, the F bit of the block is set and the EP bit is reset, recording the fact that the block has been recently filled and must not be prioritized for eviction in the current interval. At the end of each interval, the EP bit of a VC entry is set to the inverse of the F bit signifying that the entry can be considered for eviction in the next interval if the F bit is reset. The F bits of all VC entries are gang-cleared at the beginning of each interval signifying the start of a new generation. Thus, a VC block becomes eligible for eviction within two consecutive intervals.

Now, we discuss the victim caching protocol. On receiving an L1 cache victim block B, the SRA-gNRF-VC design first looks for an invalid way in the target VC set. If there is no such way, it locates the way w with the lowest SRA ratio category (say, C_i) in the target set. If there are multiple ways with the lowest SRA ratio category, the ones with their EP bits set are selected and then among them the one with the lowest physical way id is selected. Let the SRA ratio category of the L1 cache victim block B currently being considered for allocation in the VC be C_j. The SRA-gNRF-VC design victimizes the entry w to allocate block B only if one of the following two conditions is met: (i) $i < j$, (ii) $i == j$ and the EP bit of w is set. The first condition helps attract a subset of high SRA ratio blocks into the VC, while the second condition creates an avenue for replacing useless VC entries of a certain SRA ratio category.

We set the generation length to the interval between the fill and a hit to a VC entry, averaged across all entries that experience hits. We dynamically estimate this interval as follows. The interval length is measured in multiples of 4K cycles and the maximum interval length that our hardware can measure is 4M cycles. The VC controller maintains a ten-bit counter T which is incremented by one every 4K cycles (measured using a twelve-bit counter P). Each VC entry is extended by ten bits to record the value of counter T whenever the entry is filled. On a hit to a VC entry, the value of counter T recorded at the time of fill in the entry (T_{fill}) is compared with the current value of counter T ($T_{current}$). If $T_{fill} < T_{current}$, the difference between $T_{current}$ and T_{fill} is added to a counter A maintained in the VC controller. The counter A records the accumulated time between a fill and a hit to a VC entry. Another counter B maintained in the VC controller records the number of values added to counter A. At any point in time, the generation length is estimated as $\frac{A}{B}$. At the beginning of an interval, this value is copied to a generation length counter (GLC), which is decremented by one every 4K cycles. A generation ends when this counter becomes zero. Both the counters A and B are halved when either of them has saturated. When counter T saturates, it is reset to zero.

Overall, fourteen state bits are required per VC entry for implementing the SRA-gNRF policy (SRA ratio category: 2 bits, T_{fill}: 10 bits, and F and EP bits) and two additional bits per L1 cache entry for maintaining the SRA ratio category. For a 64 KB VC and 32 KB instruction and data L1 caches with 64-byte blocks, this overhead is equivalent to 16K bits (2 KB) per core. The counters T, P, A, B, and GLC require a few tens of bits per core.

2) SRA-VCUB-RProb-VC: The SRA-gNRF-VC design assumes that the high SRA ratio blocks will enjoy hits in the VC. However, this may not be true in all phases of execution. Additionally, this design also loses opportunity of caching some of the lower SRA ratio blocks that may enjoy some hits in the VC. The SRA-VCUB-RProb-VC design remedies these problems by directly considering the probability that an L1 cache victim would be reused from the VC. It partitions the L1 cache victims into several categories, estimates the collective reuse probability (RProb) of each category, and caches only the L1 cache victims belonging to the categories with high enough reuse probability. Additionally, this design substitutes the gNRF policy by a more efficient replacement policy similar to the static re-reference interval prediction (SRRIP) policy [16], which has been shown to outperform the not-recently-used and least-recently-used policies for large inclusive LLCs. This policy, like gNRF, requires only two replacement state bits used to encode four possible ages of a block in the VC.

The L1 cache victims are partitioned online based on the SRA ratio categories and a simple estimate of reuse distance. The reuse distance estimate is obtained as follows. The private cache residency of a block begins when it is fetched into the L1 cache from either LLC or main memory. Its private cache residency ends when it is evicted from the VC or from the L1 cache and not admitted to the VC. During this private cache residency, the block may make multiple trips between the L1 cache and the VC. If a block enjoys at least one use in the VC, it indicates a relatively short reuse distance of the block. We use this indication as an estimate of the reuse distance of the block. This is recorded by maintaining a VC use bit (VCUB) per block in the L1 cache and the VC. When a block is fetched into the L1 cache from the LLC or the main memory, its VCUB is set to zero. The VCUB of a block becomes one when it experiences its first hit in the VC. After this, the VCUB remains set during the rest of the block's private cache residency. An L1 cache victim having VCUB=0 is estimated to have a relatively larger reuse distance and smaller reuse probability compared to an L1 cache victim with VCUB=1. The VCUB induces a top-level partitioning of the L1 cache victims.

The L1 cache victims with VCUB=0 are further partitioned into four classes based on the victims' SRA ratio categories (C_0, C_1, C_2, C_3). For each category, the collective probability of reuse in the VC is estimated online as follows. Eight sets are sampled from the VC and the accesses to these sampled sets for blocks with VCUB=0 are used to estimate the reuse probabilities. The VC controller maintains two counters for each SRA ratio category C_i. One counter (f_i) maintains the number of fills to the sampled sets for category C_i blocks with VCUB=0. The other counter (h_i) maintains the number of hits in the sampled sets experienced by the blocks with VCUB=0 and category C_i. The reuse probability p_i of category C_i given VCUB=0 is h_i/f_i. Periodically, all the eight counters are halved.

Next, we discuss the victim caching protocol of the SRA-VCUB-RProb-VC design. The following two principles guide the VC allocation policy. First, all L1 cache victims mapping to the sampled sets are allocated in the VC because the reuse probabilities are learned from the behavior of the blocks in the sampled sets. Second, the age assigned to a block allocated in the VC is zero, two, or three depending on the estimated reuse probability of the partition containing the block. A higher reuse probability is associated with a lower age, which, in turn, signifies a lower eviction priority. The L1 cache victims with VCUB=1 are assumed to have the maximum reuse probability. The dynamic reuse probability of the L1 cache victims with VCUB=0 are estimated on-the-fly, as already discussed. On receiving an L1 cache victim block B, if the VCUB of B is set, it is allocated in the VC and assigned an age zero. If the VCUB of B is reset and B maps to a sampled set, it is allocated in the VC and assigned an age two. If the VCUB of B is reset and B does not map to a sampled set, its SRA ratio category C_i decides further actions. Let the current reuse probability estimate of C_i be p_i. If p_i is at least 1/8 (implemented as $h_i \geq [f_i \text{ shifted right by 3 bit positions}]$), the block B is allocated in the VC and assigned an age two. If p_i is less than 1/8, but there is an invalid way in the target VC set, the block B is allocated in that way and assigned an age three. If p_i is less than 1/8 and there is no inavlid way, the block B is not allocated in the VC. We have experimented with four reuse probability thresholds, namely, 1/2, 1/4, 1/8, and 1/16. Among these, 1/8 is found to achieve the best performance. Tables I and II summarize the VC operations.

TABLE I
VC ALLOCATION PROTOCOL FOR AN L1 CACHE VICTIM BLOCK

Block attributes	Maps to a VC sample set	Doesn't map to a VC sample set
VCUB=1	Allocate with age=0	Allocate with age=0
VCUB=0; SRA category $C_i; p_i \geq 1/8$	Allocate with age=2; f_i++	Allocate with age=2
VCUB=0; SRA category $C_i; p_i < 1/8$		Allocate with age=3 if an invalid way is available

TABLE II
VC ACTIONS ON A HIT

Block attributes	Maps to a VC sample set	Doesn't map to a VC sample set
VCUB=1	Invalidate; copy to L1	Invalidate; copy to L1
VCUB=0; SRA category C_i	Invalidate; copy to L1; h_i++; VCUB$\leftarrow$1	Invalidate; copy to L1; VCUB$\leftarrow$1

Within a set, the VC policy evicts a block with age three; a tie among multiple such blocks is broken by victimizing the block at the lowest physical way. If no such block exists in the set, the ages of all blocks in the set are incremented until a block with age three is found.

Overall, five state bits are required per VC entry for implementing the SRA-VCUB-RProb policy (SRA ratio category: 2 bits, VCUB: 1 bit, age: 2 bits) and three extra bits per L1 cache entry (SRA ratio category: 2 bits, VCUB: 1 bit). For a 64 KB VC and 32 KB instruction and data L1 caches with 64-byte blocks, this overhead is equivalent to 8K bits (1 KB) per core. The additional overhead of the h_i and f_i counters (nine bits each) is 72 bits per core.

IV. Simulation Environment

We use an in-house modified version of the Multi2Sim simulator [35] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. The details of the baseline configuration are presented in Table III. The interconnect switch microarchitecture assumes a four-stage routing pipeline with one cycle per stage at 2 GHz clock. The stages are buffer write/route computation, virtual channel allocation, output port allocation, and traversal through switch crossbar. There is an additional 1 ns link latency to copy a flit from one switch to the next. The overall hop latency is 3 ns.

We evaluate our VC proposals for two configurations, namely, 32 KB 8-way and 64 KB 8-way. These have lookup latencies of one and two cycles, respectively. We also explore how our proposals fare against traditional non-inclusive/non-exclusive 8-way L2 caches of capacity 32 KB, 64 KB, and 128 KB exercising fill-on-miss and LRU as well as state-of-the-art replacement policies. These L2 cache configurations have lookup latencies of one, two, and three cycles, respectively. The latencies have been fixed using CACTI [12] assuming 22 nm technology node (we use the version of CACTI distributed with McPAT [13]).

TABLE III
BASELINE SIMULATION ENVIRONMENT

On-die cache hierarchy, interconnect, and coherence directory
Per-core iL1 and dL1 caches: 32 KB, 8-way, LRU, latency 1 cycle
Shared LLC: 32 MB, 16-way, 128 banks, LRU, bank lookup latency 4 cycles for tag + 2 cycles for data, non-inclusive/non-exclusive, fill on miss, no back-inval. on eviction
Cache block size at all cache levels: 64 bytes
Interconnect: 2D mesh clocked at 2 GHz, two-cycle link latency (1 ns), four-cycle pipelined routing per switch (2 ns latency); Routing algorithm: dimension-order-routing; Each switch connects to: a core, its L1 caches, one LLC bank, one 4× (relative to per-core L1 caches) sparse directory slice [11], [31].
Sparse directory slice: 16-way, LRU replacement
Coherence protocol: write-invalidate MESI
Main memory
Memory controllers: eight single-channel DDR3-2133 controllers, evenly distributed over the mesh, FR-FCFS scheduler
DRAM modules: modeled using DRAMSim2 [32], 12-12-12, BL=8, 64-bit channels, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices, open-page policy

The applications for this study are drawn from various sources and detailed in Table IV (ROI refers to the parallel region of interest). Since many-core shared memory processors are prevalently used for commercial computing, we pick seven of our nine applications from the domain of web and data serving (SPECWeb-B, SPECWeb-E, SPECWeb-S, TPC-C, TPC-E, TPC-H, SPEC JBB). Additionally, we pick one application (barnes) as a representative of scientific computing, which often exercise large-scale shared memory servers. One application (bodytrack) is selected from the domain of computer vision where parallel processing is quite popular. The inputs, configurations, and simulation lengths are chosen to keep the simulation time within reasonable limits while maintaining fidelity of the simulation results. The PARSEC and SPLASH-2 applications are simulated in execution-driven mode, while the rest of the applications are simulated by replaying an instruction trace collected through the PIN tool capturing all activities taking place in the application address space. The PIN trace is collected on a 24-core machine by running each multi-threaded application creating at most 128 threads (including server, application, and JVM threads). Before replaying the trace through the simulated 128-core system, it is preprocessed to expose maximum possible concurrency across the threads while preserving the global order at global synchronization boundaries and between load-store pairs touching the same 64-byte block.

V. Simulation Results

In this section, we present a detailed evaluation of our proposal. All results are normalized to a baseline design with 32 KB 8-way instruction and data L1 caches per core and no L2 cache. The shared LLC is 32 MB 16-way in all configurations.

A. Performance Evaluation

We begin the discussion on performance evaluation by comparing the four VC designs presented in Section III. Figure 9 quantifies the percentage reduction in core cache misses relative to the baseline for the four VC designs, namely, LRF-VC, NRF-VC, SRA-gNRF-VC, and SRA-VCUB-RProb-VC. We have also included the results for an optimal VC design that implements Belady's optimal replacement algorithm [5], [30] extended with the option of not allocating a block in the VC if its next-use distance is larger than all blocks in the target set. The optimal design requires knowledge about the future accesses. It is evaluated offline after collecting the access trace for each application. All VC designs have 64 KB 8-way configuration. On average, both LRF and NRF reduce the core cache misses by 40%, while SRA-gNRF achieves a 41.3% reduction. The SRA-VCUB-RProb design achieves a reduction of 44.1% having a less than 8% gap to the optimal design, which achieves a reduction of 51.9%. Compared to LRF and NRF, the top gainers of the SRA-VCUB-RProb design include bodytrack, barnes, and TPC-C. For TPC-H, the SRA-VCUB-RProb design achieves near-optimal core cache misses.

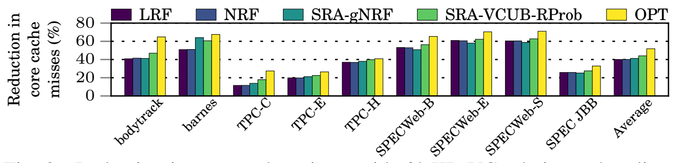

Fig. 9. Reduction in core cache misses with 64 KB VC relative to baseline.

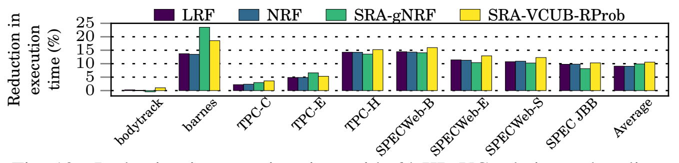

Fig. 10. Reduction in execution time with 64 KB VC relative to baseline.

Figure 10 presents the percentage reduction in execution time for the four VC designs with 64 KB capacity relative to the baseline. The performance of the optimal design cannot be evaluated because the future accesses cannot be fixed online. On average, LRF and NRF save 9% execution time, while the SRA-gNRF and SRA-VCUB-RProb designs reduce execution time by 9.9% and 10.6%, respectively. Within each application, the savings in execution time correspond well to the relative trend shown in Figure 9. Bodytrack fails to improve much in performance because saving core cache misses is not particularly important for its performance.

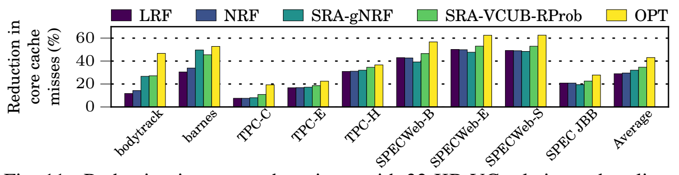

Fig. 11. Reduction in core cache misses with 32 KB VC relative to baseline.

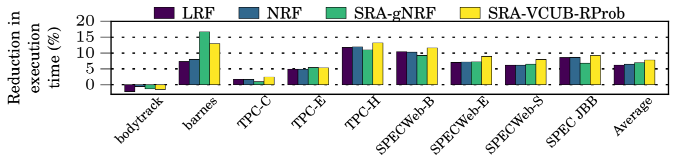

Fig. 12. Reduction in execution time with 32 KB VC relative to baseline.

Figures 11 and 12 evaluate the VC designs with 32 KB capacity. All designs continue to be 8-way set-associative. On average, the SRA-VCUB-RProb design saves 35% core cache misses relative to the baseline and is only 8% away from the optimal design, which saves 43% core cache misses. Compared to LRF and NRF, bodytrack and barnes continue to enjoy large savings in core cache misses with the SRA-VCUB-RProb design. The SRA-VCUB-RProb design achieves a 7.9% reduction in execution time compared to the baseline, on average. We will consider only the best performing VC design i.e., SRA-VCUB-RProb in further evaluation.

Next, we compare the SRA-VCUB-RProb design with the traditional non-inclusive L2 caches exercising LRU replacement policy. Figure 13 shows the percentage reduction in core cache misses relative to the baseline for 32 KB and 64 KB SRA-VCUB-RProb-VC design and 32 KB, 64 KB, and 128 KB traditional non-inclusive L2 cache design. On average, the 32 KB traditional L2 cache design saves only 6% core cache misses, while the 32 KB VC design saves an impressive 35% core cache misses; the 64 KB traditional L2 cache design saves

TABLE IV
SIMULATED APPLICATIONS

Suite	Applications	Input/Configuration	Simulation length
PARSEC	bodytrack	sim-medium	Complete ROI
SPLASH-2[3]	barnes	32K particles	Complete ROI
TPC	MySQL TPC-C	10 GB database, 2 GB buffer pool, 100 warehouses, 100 clients	500 transactions
	MySQL TPC-E	10 GB database, 2 GB buffer pool, 100 clients	Five billion instructions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 100 clients, zero think time, even distribution of Q6, Q8, Q11, Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.2	Banking (SPEC Web-B), Ecommerce (SPEC Web-E), Support (SPEC Web-S); Worker thread model, 128 simultaneous sessions, mod_php module	Five billion instructions
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions

[3] The SPLASH-2 applications are drawn from the SPLASH2X extension of the PARSEC distribution.

22.3% core cache misses, while the 64 KB VC design saves 44.1% core cache misses. Most importantly, a 128 KB traditional L2 cache design saves 42.2% core cache misses. This saving is a couple of percentages *lower* than what a half-sized (64 KB) VC exercising the SRA-VCUB-RProb design achieves. Similarly, a 32 KB VC saves much higher percentage of core cache misses than a traditional 64 KB L2 cache. Compared to 128 KB traditional L2 cache, our 64 KB SRA-VCUB-RProb-VC design saves 7.78 MB of on-die SRAM storage for our 128-core configuration assuming 48-bit physical address (our proposal's overheads of VC/L1 cache state bits and additional 12 bits per directory entry are accounted for).

Figure 14 compares the SRA-VCUB-RProb-VC design with the traditional L2 caches in terms of percentage saving in execution time relative to the baseline. Across the board, the VC design outperforms the same-sized traditional L2 caches by large margins. More importantly, a 32 KB VC outperforms a 64 KB traditional L2 cache and a 64 KB VC outperforms a 128 KB traditional L2 cache, on average. These results strongly advocate the replacement of traditional private L2 caches by specialized per-core VC designs that can achieve significant space saving per core while delivering better performance at lower interconnect traffic.

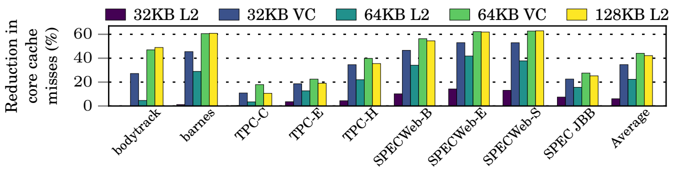

Fig. 13. Comparison between traditional non-inclusive L2 cache and SRA-VCUB-RProb-VC in terms of reduction in core cache misses.

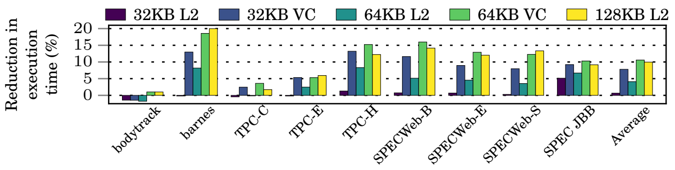

Fig. 14. Comparison between traditional non-inclusive L2 cache and SRA-VCUB-RProb-VC in terms of reduction in execution time.

Figures 15 and 16 compare our SRA-VCUB-RProb-VC design with iso-capacity baselines. A configuration with a 32 KB VC invests a total of 96 KB cache per core when the 32 KB L1 instruction and data cache capacities are included. This configuration should be compared against a baseline that also invests 96 KB of L1 caches per core. So, we evaluate a configuration that has 48 KB 6-way L1 instruction and data caches per core. Similarly, we compare the 64 KB VC design with a configuration that has 64 KB 8-way L1 instruction and data caches per core. The 48 KB L1 caches have a single-cycle lookup latency, while the 64 KB L1 caches have two-cycle lookup latency. The results are shown relative to the baseline with 32 KB L1 instruction and data caches per core. Figures 15 and 16 respectively show that the 48 KB L1 cache configuration saves 20% core cache misses and 4.6% execution time, while the 32 KB VC saves 35% core cache misses and 7.9% execution time, on average. Both 32 KB and 64 KB VC designs save more core cache misses and execution time than the 64 KB L1 cache configuration, on average. The 64 KB L1 cache configuration saves 32.5% core cache misses and 6.3% execution time, while the 64 KB VC saves 44.1% core cache misses and 10.6% execution time, on average. Overall, for all applications, the SRA-VCUB-RProb-VC design comfortably outperforms the iso-capacity baseline configurations.

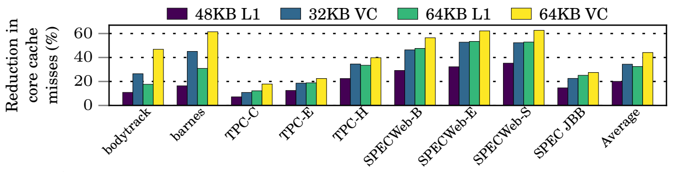

Fig. 15. Comparison between iso-capacity L1-only baselines and the SRA-VCUB-RProb-VC configurations in terms of reduction in core cache misses.

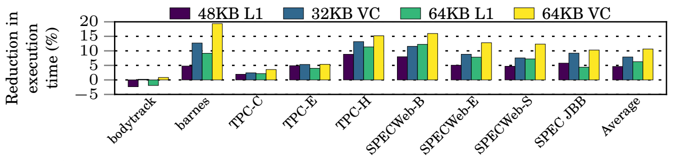

Fig. 16. Comparison between iso-capacity L1-only baselines and the SRA-VCUB-RProb-VC configurations in terms of reduction in execution time.

A cost-effective alternative to designing a specialized VC is to enhance the L1 caches in a traditional single-level private cache hierarchy and the L2 cache in a traditional two-level private cache hierarchy with state-of-the-art replacement/insertion policies. We explore how the configurations equipped with the SRA-VCUB-RProb-VC design fare against the traditional single-level (L1 cache only) and two-level (L1 and L2 caches) private cache hierarchies enhanced with the signature-based hit prediction (SHiP) policy proposal [37]. The instruction L1 and the data L1 caches of the traditional single-level private cache hierarchy are enhanced with the SHiP-mem and SHiP-PC policies, respectively. The L2 cache in the traditional two-level private cache hierarchy is enhanced with SHiP-mem for instruction blocks and SHiP-PC for data blocks. We compare these configurations against the SRA-VCUB-RProb-VC design working with the traditional baseline 32 KB L1 caches exercising the LRU replacement policy. Figures 17 and 18 show the results normalized to the baseline with 32 KB L1 instruction and data caches per core exercising the LRU replacement policy. The single-level private cache hierarchy configurations with L1 caches exercising SHiP policy are shown as 32 KB L1, 48 KB L1, and 64 KB L1. The two-level private cache hierarchy configurations with non-inclusive L2 cache exercising SHiP policy are shown as 32 KB L2, 64 KB L2, and 128 KB L2. The 32 KB and 64 KB VC configurations use the SRA-VCUB-RProb design along with 32 KB instruction and data L1 caches exercising LRU policy.

A comparison of Figures 17 and 18 with Figures 15 and 16 shows that the SHiP policy is not particularly effective for L1 caches and perform close to the LRU policy. On the other hand, a comparison with Figures 13 and 14 shows that the SHiP policy when implemented in the L2 cache is able to improve the performance by a reasonable amount relative to LRU policy, on average. The SHiP policies are designed to work well for caches that experience an access stream

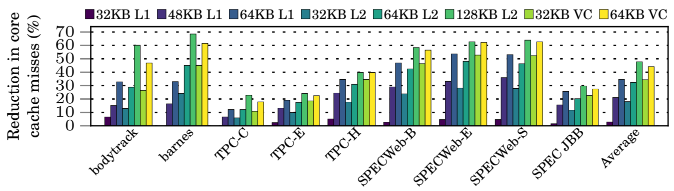

Fig. 17. Comparison of core cache miss savings between the SRA-VCUB-RProb-VC design and single-level and two-level private hierarchy configurations enhanced with the SHiP policy.

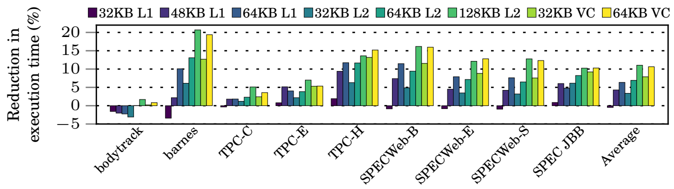

Fig. 18. Comparison of execution time savings between the SRA-VCUB-RProb-VC design and single-level and two-level private hierarchy configurations enhanced with the SHiP policy.

with filtered locality e.g., L2 and outer level caches. Nonetheless, Figures 17 and 18 show that both 32 KB and 64 KB SRA-VCUB-RProb-VC designs save more core cache misses and execution time than all traditional configurations enhanced with the SHiP policies except the 128 KB L2 cache configuration, on average. The 128 KB L2 cache working with the SHiP policies outperforms the 64 KB VC design only marginally (11.0% vs. 10.6% execution time saving) and saves only a few percentage extra core cache misses (47.8% vs. 44.1%), on average. Overall, the SRA-VCUB-RProb-VC design continues to outperform the iso-capacity single-level as well as two-level private cache hierarchies enhanced with the SHiP policies.

In summary, our detailed performance evaluation presents a compelling case for the SRA-VCUB-RProb-VC design as the private per-core L2 cache in many-core server processors. It comfortably outperforms the iso-capacity traditional single-level and two-level non-inclusive private cache hierarchy designs exercising LRU as well as SHiP policies. Further, a 32 KB VC outperforms a 64 KB traditional L2 cache and a 64 KB VC outperforms a 128 KB traditional L2 cache exercising the LRU policy presenting an opportunity to halve the L2 cache space per core with better performance. When the traditional L2 cache is enhanced with the SHiP policy, a 128 KB L2 cache marginally outperforms our 64 KB SRA-VCUB-RProb-VC design.

B. Interconnect Traffic Comparison

Figure 19 compares the on-die interconnect traffic (total number of bytes transferred) for five different private cache configurations. For each application, the leftmost two bars represent single-level private cache configurations with 32 KB and 64 KB L1 caches exercising LRU policy and no L2 cache. The next two bars represent configurations with 64 KB and 128 KB non-inclusive L2 cache exercising LRU policy. The rightmost bar represents a configuration with 64 KB VC exercising our SRA-VCUB-RProb policy. The last three configurations exercise 32 KB L1 caches with LRU policy. The interconnect traffic for each configuration is divided into four categories. The private cache misses and their responses constitute the processor traffic. The private cache evictions to the LLC and their acknowledgements constitute the writeback traffic. The requests and replies to and from the memory controllers constitute the DRAM traffic. Everything else constitutes the coherence traffic. The results are normalized to the total interconnect traffic of the leftmost bar in each application. Across the board, our VC proposal saves significant portions of the interconnect traffic arising primarily from the savings in processor misses and private cache evictions. Compared to the baseline 32 KB L1 configuration (leftmost bar), our proposal saves 43.1% interconnect traffic. Compared to the 64 KB L1 and 64 KB L2 configurations, our proposal saves 11.5% and 20.9% interconnect traffic, respectively. These two configurations invest the same total cache (128 KB) to the per-core private cache hierarchy as our proposal does using the 64 KB VC. It is encouraging to note that compared to the 128 KB L2 configuration, our proposal saves 1.9% interconnect traffic while requiring only half the L2 cache space.

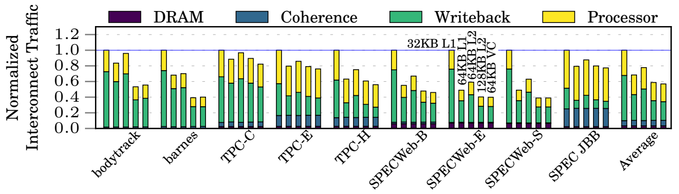

Fig. 19. Interconnect traffic for private cache hierarchy configurations.

C. Iso-capacity Energy Comparison

Figure 20 quantifies the total energy (dynamic and leakage together) expended by the on-die cache hierarchy and the sparse directory for three different iso-capacity (128 KB per core) private cache hierarchy configurations assuming 22 nm technology nodes (determined with CACTI). For each application, the leftmost bar (marked L1) shows the total energy when the per-core private cache hierarchy has 64 KB 8-way L1 instruction and data caches and no L2 cache. The middle bar (marked VC) shows the total energy when the per-core private cache hierarchy has 32 KB 8-way L1 instruction and data caches and a 64 KB 8-way VC exercising the SRA-VCUB-RProb design. The rightmost bar (marked L2) shows the total energy when the per-core private cache hierarchy has 32 KB 8-way L1 instruction and data caches and a 64 KB 8-way traditional non-inclusive L2 cache exercising LRU replacement policy. All L1 caches exercise LRU replacement policy. Each bar shows the energy contributed by the instruction L1 cache (IL1), data L1 cache (DL1), L2 cache (L2), LLC, and the coherence directory. All results are normalized to the total energy of the leftmost bar. On average, the VC configuration expends 1% less energy compared to the L1 configuration, while the L2 configuration expends 3% more energy than the L1 configuration. As a result, the 64 KB VC configuration has 6% and 10% less energy-delay product compared to the iso-capacity L1 and L2 configurations, respectively.

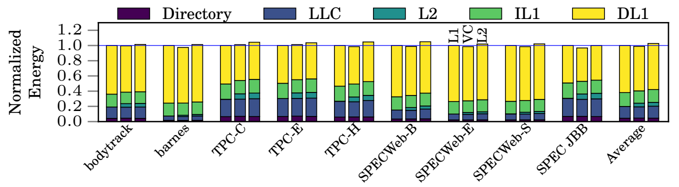

Fig. 20. Total energy expended by the cache hierarchy for various private cache hierarchy configurations with a per core private cache hierarchy budget of 128 KB.

VI. Related Work

Fully-associative very small (8 to 32 entries) victim caches were introduced to handle conflict misses in small direct-mapped caches [17]. The existing selective victim caching proposals explore run-time strategies for identifying the L1 cache evictions resulting from conflicts and capture these evictions in the fully-associative L1 victim cache of single-core processors [7], [14]. In contrast, our proposal targets larger set-associative victim caches working with L1 caches having high set-associativity where only conflict miss selection is not enough. To the best of our knowledge, this is the first proposal on sharing-aware per-core victim caching for server workloads.

The design of a large victim cache with selective insertion based on frequency of misses has been explored and is shown to work well with large inclusive LLCs [4]. Another proposal exploits the dead blocks in a large inclusive LLC to configure an "embedded" victim cache [22]. These designs are not suitable for per-core mid-level victim caches.

Our victim cache, by design, introduces an exclusive L2 cache in the per-core private cache hierarchy of the many-core processor. The advantages and disadvantages of exclusive LLCs compared to their inclusive counterparts have been explored by several studies [18], [38]. Bypass and insertion policy optimizations for large exclusive LLCs have also been studied [6], [10]. In contrast, we architect a mid-level exclusive victim cache per core in a many-core server processor.

The use of large private caches working with an exclusive LLC has been explored for a small-scale (16 cores) server processor [15]. Our proposal leaves the LLC unchanged and architects a space-efficient private cache hierarchy. Also, specialized coherence protocols that allow selective private caching of certain data based on temporal and spatial locality estimates have been proposed [26].

Several studies have explored specialized architectures for the on-die interconnect with the goal of optimizing the latency observed and energy expended in ferrying information between the private cache hierarchy and the shared last-level cache in server processors [23], [27], [28], [29], [36]. The design innovations in these studies involve the following: predicting the useful words within a requested cache block and transmitting only the flits that contain these words [23]; architecting specialized topologies and switches to accelerate instruction delivery to the cores' instruction caches [27]; designing a hybrid of virtual cut-through and circuit switched routing protocols to improve the communication latency [28]; selectively eliminating the per-hop resource allocation delay through proactive resource allocation policies [29]; designing separate request and response networks to suit the different demands of request and response packets [36]. In contrast, our proposal uses a traditional interconnection network and optimizes the interconnect traffic by designing a specialized private cache hierarchy.

A significant body of research has recognized the importance of optimizing the instruction cache performance of the cores in the context of commercial server workloads. One set of studies observes that there is a large amount of overlap within and across transactions in terms of instruction footprint and database operations in online transaction processing (OLTP) workloads. These studies exploit this instruction locality by judicious scheduling and migration of transaction threads and database actions [1], [2], [3], [34]. Another class of proposals has designed specialized instruction prefetchers [8], [9], [19], [20], [21], [24], [25]. While prefetchers can hide the inefficiencies of the private cache hierarchy, they cannot save interconnect traffic. Our proposal, instead, focuses on the design of the private cache hierarchy to improve the interconnect traffic as well as performance. The application domain of our proposal is not limited to just OLTP or instruction delivery for commercial workloads.

A recent proposal on ultra-low-overhead coherence tracking has exploited the observation that a small fraction of LLC blocks experience frequent read-sharing [33]. We show that a small fraction of the shared blocks contribute to a large fraction of core cache misses and design victim caches to capture a subset of these blocks.

VII. Summary

We have presented the designs of a victim cache working with the per-core private L1 caches of a many-core server processor. The victim cache effectively replaces the traditional private L2 cache. Our best victim caching proposal partitions the L1 cache victim space into different classes based on the degree of sharing and an indirect estimate of the reuse distance. It estimates the reuse probability of each partition dynamically and uses these estimates to decide the partitions that should be retained in the victim cache. This selective victim caching proposal, on average, saves 44.1% core cache misses and 10.6% execution time compared to a baseline that does not have a private L2 cache. Further, this proposal comfortably outperforms iso-capacity traditional single-level and two-level private cache hierarchy designs. Most importantly, 32 KB and 64 KB victim caches outperform traditional 64 KB and 128 KB LRU L2 caches, respectively. This opens up the opportunity of halving the L2 cache space investment per core while offering better performance.

References

[1] I. Atta, P. Tozun, X. Tong, A. Ailamaki, and A. Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *ISCA* 2013.

[2] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *MICRO* 2012.

[3] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos. Reducing OLTP Instruction Misses with Thread Migration. In *DaMoN* 2012.

[4] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *MICRO* 2007.

[5] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[6] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *PACT* 2012.

[7] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *MICRO* 1999.

[8] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO* 2011.

[9] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Instruction Fetch Streaming. In *MICRO* 2008.

[10] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *ISCA* 2011.

[11] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *ICPP* 1990.

[12] HP Labs. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Available at http://www.hpl.hp.com/research/cacti/.

[13] HP Labs. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. Available at http://www.hpl.hp.com/research/mcpat/.

[14] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *ISCA* 2002.

[15] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely Jr., and J. S. Emer. High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. In *HPCA* 2015.

[16] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *ISCA* 2010.

[17] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *ISCA* 1990.

[18] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *ISCA* 1994.

[19] P. Kallurkar and S. R. Sarangi. pTask: A Smart Prefetching Scheme for OS Intensive Applications. In *MICRO* 2016.

[20] C. Kaynak, B. Grot, and B. Falsafi. Confluence: Unified Instruction Supply for Scale-out Servers. In *MICRO* 2015.

[21] C. Kaynak, B. Grot, and B. Falsafi. SHIFT: Shared History Instruction Fetch for Lean-core Server Processors. In *MICRO* 2013.

[22] S. M. Khan, D. A. Jimenez, D. Burger, and B. Falsafi. Using Dead Blocks as a Virtual Victim Cache. In *PACT* 2010.

[23] H. Kim, B. Grot, P. V. Gratz, and D. A. Jimenez. Spatial Locality Speculation to Reduce Energy in Chip-Multiprocessor Networks-on-Chip. In *IEEE TC*, March 2014.

[24] A. Kolli, A. G. Saidi, and T. F. Wenisch. RDIP: Return-address-stack Directed Instruction Prefetching. In *MICRO* 2013.

[25] R. Kumar, C-C. Huang, B. Grot, and V. Nagarajan. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *HPCA* 2017.

[26] G. Kurian, O. Khan, and S. Devadas. The Locality-aware Adaptive Cache Coherence Protocol. In *ISCA* 2013.

[27] P. Lotfi-Kamran, B. Grot, and B. Falsafi. NOC-Out: Microarchitecting a Scale-Out Processor. In *MICRO* 2012.

[28] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. An Efficient Hybrid-Switched Network-on-Chip for Chip Multiprocessors. In *IEEE TC*, **65**(5): 1656–1662, May 2016.

[29] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. Near-Ideal Networks-on-Chip for Servers. In *HPCA* 2017.

[30] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[31] B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-efficient Directory Methods. In *ISCA* 1990.

[32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE CAL*, January-June 2011.

[33] S. Shukla and M. Chaudhuri. Tiny Directory: Efficient Shared Memory in Many-core Systems with Ultra-low-overhead Coherence Tracking. In *HPCA* 2017.

[34] P. Tozun, I. Atta, A. Ailamaki, and A. Moshovos. ADDICT: Advanced Instruction Chasing for Transactions. In *PVLDB* 2014.

[35] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT* 2012.

[36] S. Volos, C. Seiculescu, B. Grot, N. K. Pour, B. Falsafi, and G. De Micheli. CCNoC: Specializing On-Chip Interconnects for Energy Efficiency in Cache-Coherent Servers. In *NOCS* 2012.

[37] C-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *MICRO* 2011.

[38] Y. Zheng, B. T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. In *ISPASS* 2004.