DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors

Nadja Holtryd, Madhavan Manivannan, Per Stenström, Miquel Pericàs
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email: {holtryd, madhavan, per.stenstrom, miquelp}@chalmers.se

***Abstract*—Cache partitioning in tile-based CMP architectures is a challenging problem because of i) the need to determine capacity allocations with low computational overhead and ii) the need to place allocations close to where they are used, in order to reduce access latency. Although, previous solutions have addressed the problem of reducing the computational overhead and incorporating locality-awareness, they suffer from the overheads of *centrally* determining allocations.**

In this paper, we propose DELTA, a novel *distributed* and locality-aware cache partitioning solution which works by exchanging asynchronous challenges among cores. The distributed nature of the algorithm coupled with the low computational complexity allows for frequent reconfigurations at negligible cost and for the scheme to be implemented directly in hardware. The allocation algorithm is supported by an enforcement mechanism which enables locality-aware placement of data. We evaluate DELTA on 16- and 64-core tiled CMPs with multi-programmed workloads. Our evaluation shows that DELTA improves performance by 9% and 16%, respectively, on average, compared to an unpartitioned shared last-level cache.

***Index Terms*—cache partitioning, multicore architectures, performance isolation**

I. Introduction

Efficient use of cache resources on chip multiprocessors (CMPs) is necessary in order to bridge the speed gap between processor and main memory. The last-level cache (LLC) is usually shared among all cores to maximize utilization. Unconstrained sharing, however, can result in destructive interference between workloads and lead to large performance variation, degrade throughput and violate per-application Quality of Service (QoS) requirements. Cache partitioning can mitigate destructive interference by isolating cache space between cores/applications.

A partitioning solution typically comprises two components: an *allocation policy*, to decide the size of the partitions for each application, and an *enforcement mechanism* to enforce the partitions. With regard to the allocation policy, known approaches target different objectives, e.g. to maximize throughput or improve fairness [1]–[3]. Utility-based cache partitioning (UCP) [1] aims to maximize throughput by assigning cache ways to applications that benefit most from the cache capacity. To do so, UCP leverages the Lookahead algorithm. The algorithm determines dynamic cache allocation based on marginal utility and partitions ways in a monolithic cache between applications but it has a high computational complexity. Approaches to determine cache allocations with lower computational overhead have been proposed [4]. However, as our evaluation of the overheads (described in Section IV) shows, the scalability of these proposals remains limited by their reliance on a centralized allocation algorithm. This reliance presents two problems. First, the execution time of the allocation algorithm limits the frequency of reconfiguration, especially as we scale to large core counts. And second, the invocation of the algorithm introduces unpredictable jitter in application execution. OS noise (jitter) has been identified as a major cause of both execution time unpredictability and untimely synchronization [5], [6]. This is particularly bad for multithreaded applications that rely on bulk synchronous parallelism (BSP) [6], [7]. As a consequence, centralized allocation approaches cannot be utilized when scaling to large core counts and requiring frequent reconfigurations.

Different enforcement mechanisms for cache partitioning have been proposed. Way and set partitioning are proposed in the context of monolithic caches with few cores [1], [2], [8], [9]. These schemes have the drawback of only supporting a limited number of coarse-grained partitions. Solutions have been proposed to enable fine-grained partitioning [10]–[17]. The main shortcoming of these techniques is that they do not take locality into account when partitioning LLCs in tiled CMPs. A few proposals have tried to address this limitation by enabling locality-aware placement [4], [18]. However, the allocation policies used in these proposals rely on a centralized allocation component and inherit its shortcomings.

An ideal cache partitioning solution should be fine-grained to support many and varying partitions, be locality-aware to place data close to where it is used, and adapt quickly to changes in application-phase behavior while still ensuring that allocation operations can be performed in a scalable manner, with low overhead and minimal OS intervention. We propose DELTA, a novel scalable cache partitioning solution for tile-based CMPs, that utilizes a *distributed* allocation policy and a locality-aware enforcement mechanism. In contrast to prior work, our solution uses a completely distributed and asynchronous allocation algorithm, works with a standard LRU-replacement policy, does locality-aware enforcement and is virtually transparent to the full software stack.

DELTA's Allocation Policy: DELTA's allocation algorithm comprises an inter-bank and an intra-bank component. The inter-bank algorithm determines capacity allocations by asynchronously exchanging *challenges* among cores. A challenge represents the performance benefit of obtaining increased cache capacity and helps an application with larger performance potential to gain more cache capacity. The algorithm uses coarse-grained shadow-tags to collect reuse-distance information with minimal overhead, which is used as the basis for computing a challenge. The intra-bank algorithm periodically redistributes the cache capacity within a bank by giving more space to the application that has a larger potential performance gain. The distributed nature of the algorithm enables quick and incremental adaptation to program phase changes without requiring a central entity to determine and perform chip-wide reallocation of cache capacity for the different applications.

DELTA's Enforcement Mechanism: DELTA combines way partitioning and bank-level partitioning to achieve a locality-aware and fine-grained partition-enforcement mechanism. The partitioning solution uses per-core *Cache Bank Tables* (CBTs), where mappings between addresses and banks are recorded. When a request needs to access the LLC, the CBT is used to identify the cache bank that the address is mapped to. Inside each bank, way partitioning is used to divide the capacity. The flexibility of the enforcement mechanism makes it possible to keep data close to where it is used.

In summary, we make the following contributions:

(a) We propose DELTA, a fully distributed and locality-aware cache-partitioning solution. The distributed allocation algorithm asynchronously negotiates and makes effective cache allocation decisions. The flexibility of the enforcement mechanism enables locality-aware mappings. The distributed nature of the solution, coupled with low computational overhead, enables a hardware-based implementation. This allows the scheme to scale to large core counts while permitting frequent reconfigurations without invoking the operating system.

(b) We describe a novel allocation policy consisting of a coarse-grained and a fine-grained component which are responsible for carrying out inter- and intra-bank allocations, respectively. The inter-bank algorithm uses challenges to expand into multiple banks, while the intra-bank algorithm allows the allocation to grow within a bank.

(c) We present a reconfigurable NUCA enforcement mechanism that enables locality-aware mapping. The two-level mechanism combines coarse-grained, bank-level partitioning with fine-grained way partitioning. The CBT enforces flexible mapping of addresses to cache banks, which enables placing data close to where it is used.

We evaluate our solution on 16- and 64-core tiled CMPs. With multi-programmed workloads on a 16-core CMP we obtain speed-ups of up to 16% (geom. mean 9%) compared to an unpartitioned S-NUCA and up to 11% (geom. mean 6%) compared to private caches (equal partitioning). On the 64-core CMP DELTA improves performance by up to 28% (geom. mean 16%) over an unpartitioned S-NUCA.

The rest of the paper is organized as follows: Section II describes our proposed solution in detail. Section III discusses the methodology and Section IV presents the evaluation of the proposal. We provide an overview of related work in Section V and conclude in Section VI.

II. DELTA Cache Partitioning

Section II-A provides an overview of DELTA, followed by a detailed presentation of the algorithms and mechanisms in subsequent sections.

A. Overview

Both the allocation policy and the enforcement mechanism have an $inter$-$bank$ and an $intra$-$bank$ component. The inter-bank component takes care of the allocation and enforcement across cache banks, and the intra-bank part handles the allocation and enforcement within the banks.

DELTA allocation policy: Figure 1 provides an overview of the distributed allocation algorithm. The inter-bank allocation algorithm works by tiles periodically sending out challenge messages, as shown in Figure 1 (Step #1). The mechanism relies on two metrics called *pain* and *gain* (see Table I). A challenge message contains the potential *gain* that a given application would experience if it were to get additional cache capacity. The challenged tile compares its own *pain*, owing to predicted decrease in performance because of lost cache space, with the gain (Step #2), (see Section II-B2 for details about pain and gain). If the gain is greater, the challenged tile gives up space in the cache bank and informs the challenger tile with a response message (Step #3). A portion of the addresses ($[A_k - A_l]$ in Figure 1) belonging to the application running in the challenger tile is remapped to the cache bank in the challenged tile (Step #4). The addresses that have been remapped to a different bank are then invalidated in their previous location. The inter-bank allocation policy helps applications acquire additional cache capacity by mapping data to cache banks in other tiles.

The second part of the allocation algorithm, the intra-bank algorithm, governs changes within a bank. The intra-bank algorithm is invoked periodically in every cache bank. In each interval some ways are transferred to the partition with most gain from the partition with the least. This way, reassignment has little overhead since it does not affect the mapping of addresses to banks, and therefore does not lead to invalidations. While the inter-bank allocation algorithm helps an application to expand its working set into other tiles and get a fixed capacity, the intra-bank algorithm helps in fine-tuning the capacity in banks that an application has already expanded

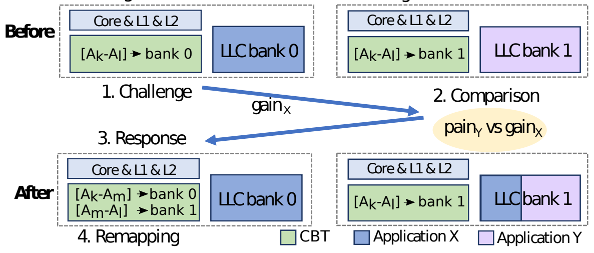

Fig. 1: Overview of steps in DELTA allocation.

Gain	Predicted performance increase due to increased cache space
Pain	Predicted performance decrease due to lost cache space

TABLE I: Terminology

into. The intra-bank algorithm reports information about ways that an application wins/loses in a cache bank outside the tile and this acts as a feedback to guide inter-bank expansion. The details about the DELTA allocation algorithm are discussed in Section II-B.

DELTA enforcement mechanism: The inter-bank mechanism uses a mapping table as shown in Figure 1, the CBT, to map addresses to cache banks (see Section II-C for details). On a private cache miss the CBT provides the mapping of each address to the LLC bank where it resides. The mappings in the CBT are changed when reconfigurations are triggered by the allocation algorithm. The flexible mapping enabled by the CBT is in contrast to S-NUCA which maps addresses to cache banks statically. The intra-bank mechanism relies on hardware support for way partitioning (discussed in Section II-C) similar to that available in some commodity systems [19].

B. DELTA Allocation

1) Allocation Algorithm: Inter-bank allocation: The pseudo-code for the inter-bank allocation algorithm is shown in Algorithm 1. Each tile (Challenger) starts by computing the pain and gain at the beginning of every inter-bank reconfiguration interval (i_{inter}), which is set to 1ms. An analysis motivating this interval is presented in Section IV-D. A challenge message is only sent if the calculated gain is above a threshold and the size of the allocation is larger than the minimum limit (line 4). The requirement to be above the minimum allocation limit is to avoid placing data far away instead of expanding in the home bank. The choice of which tile to challenge (Challenged) is based on the distance to the tile. Each tile will start by challenging the closest neighbouring tiles, with a hop distance of one, before choosing tiles further away (line 5). A single challenge is issued by every tile in each (i_{inter}) interval if it satisfies the preconditions. The algorithm will only pick a particular tile for a second challenge after it has exhausted other candidates, regardless of whether the previous attempt was successful.

When a challenge is received, the gain from the challenger tile is compared to the pain of the challenged tile. The algorithm uses pain, instead of gain, for comparison in order to accommodate the potentially high impact on performance for the application running on the challenged tile. Furthermore, it also acts as a deterrent to prevent one tile from easily invading and taking over the capacity of neighbouring tiles. In case an application running on tile A is sharing its cache bank with another running on tile B and receives a challenge from a different application running on tile C, the algorithm will compare the Pain_{A}, Gain_{B} and Gain_{C} to determine if the challenge is successful (line 10). In case the challenge is successful, a fixed capacity (number of ways) is allocated in the challenged tile and a response is sent to the challenger tile as a notification (line 12-13). On receiving a successful

Algorithm 1: Inter-bank allocation pseudo-code

```
Input: mlp, allocationForChallanger, interDeltaWays
1  In tile Challanger at time period $i_{inter}$ ;
2  pain = calculatePain($mlp$, $allocationForChallanger$);
3  rawGain = calculateRawGain($mlp$,
     $allocationForChallanger$);
4  if rawGain >gainThreshold AND allocationForChallanger
     >minWays then
5      challenged = getClosestNeighbour();
6      gain = rawGain / distanceTo($challenged$);
7      challenge($challenged$, $challenger$, $gain$);
8  end
9  In tile Challenged on receiving a challenge;
10 $partition$ = partitionWithSmallestGainOrPainInChallenged(
     $challengedPain$, $challengerGain$,
     $gainChallengedPartitions$);
11 if partition then
12     updateWayPartition($challenger$, $partition$,
         $interDeltaWays$);
13     respondWithNewPartition($challenged$, $challenger$,
         $true$);
14 else
15     respondWithNewPartition($challenged$, $challenger$,
         $false$);
16 end
17 On response;
18 if success then
19     updateCacheBankTableWithNewPartitionIn($challenged$);
20     invalidateAddressesWithChangedBankPlacement();
21 end
22 markAsChallenged($challenged$);
```

response the CBT is updated and invalidations are triggered if required (line 18-20). The intra-bank algorithm, described later in this section, discusses how the allocation for the challenger tile can grow to encompass the entire cache bank gradually over time. If the challenged tile does not use its home cache bank (i.e. the core is idle), the algorithm will allocate the whole cache bank to the challenger tile immediately instead of gradually. This is done to make it easier for applications that are running alone to increase their allocation quickly as the cache banks will otherwise remain underutilized.

Intra-bank allocation: The pseudo-code for the intra-bank algorithm is shown in Algorithm 2. This is triggered in each tile at every intra-bank reconfiguration interval (i_{intra}) which is set to 0.1ms. The algorithm works by comparing the gain for each partition that shares the cache bank (line 2-3) and reassigns some ways ($intraDeltaWays$) from the partition that has the least gain to the one that has the most (line 5). Here, unlike the inter-bank allocation, the comparison only considers the gain of every application to determine the winner, for two reasons. Firstly, the application running on the tile must have already demonstrated a significant gain, more than the home bank's pain, to have been allowed to expand into a different tile. Secondly, intra-bank changes in allocation are lightweight and do not introduce any invalidation-related overheads (except when leaving a tile). In case an application running on tile A is sharing its own cache bank with two others running on tile B and C, a comparison will happen between Gain_{A}, Gain_{B} and Gain_{C}, to determine which contending

Algorithm 2: Intra-bank allocation pseudo-code

```
  Input: $intraDeltaWays$
1 In each tile at time period $i_{intra}$;
2 $partitionSmallest$ =
    partitionWithSmallestGain($gainsOfPartitionsInBank$,
    $minWays$);
3 $partitionLargest$ =
    partitionWithLargestGain($gainsOfPartitionsInBank$);
4 if $partitionWithLargestGain$ !=
  $partitionSmallestGain$ then
5     updateWayPartitioning($partitionLargest$,
        $partitionSmallest$, $intraDeltaWays$);
6     reportNewAllocation($partitionSmallest$,
        $partitionLargest$);
7 end
```

application will win/lose cache ways.

After reassignment the information about the number of ways are sent back to the respective contending home tiles (line 6). This acts as a feedback mechanism between the intra- and inter-bank algorithm since the current allocation is an important factor in determining the pain and gain, as will be described in the next section. The inter- and intra-bank algorithms are invoked periodically after an initial state where cache capacity is equally partitioned among all tiles.

2) Computing Gain and Pain: The measure of pain and gain is based on simple heuristics that lead to effective allocation decisions. We use gain to predict how an application will react to an increase in cache capacity ($gainWays$) by expanding in/to other tiles. In order to compute gain we take into consideration information about the number of misses provided by the shadow tags, memory-level parallelism (MLP), current capacity allocation outside the home tile and the hop distance. The potential gain for an application running on tile i and expanding into tile j is calculated using the following formula:

$$Gain_{\text{i,j,gainWays}} = \frac{a_{gainWays} * (k+1)^{-1}}{m * (l+1)} \quad (1)$$

where a is the number of misses that potentially can be avoided with $gainWays$ additional ways, k is the number of ways outside of the home tile, m is the MLP of the application running on tile i and l is the hop distance from tile i to j.

The rationale behind factoring in the aforementioned attributes in the gain expression is as follows. Firstly, factoring in the number of avoidable misses provides an estimate of reduction in the number of long-latency memory accesses which influences performance. This value can be read directly from the shadow tags in the monitoring hardware for a given core. MLP is factored in because this coupled with the number of misses helps to get a better estimate of the performance impact of cache allocation decisions. The MLP estimate is obtained through performance counters. Lastly, we factor in the current allocation in remote tiles and the hop distance to introduce fairness and ensure that no single application expands its allocation too aggressively.

We use pain as a heuristic measure to predict how an application will react to losing available cache capacity ($painWays$) on the home tile where it is running. The pain value is never communicated to other cache banks. In order to compute pain we only take information about misses provided by the shadow tags and MLP into account. Unlike the formula for gain, we do not take information about allocations outside home bank and the distance into account because the goal here is to protect the capacity allocation in the home tile where the application is running. Since the pain is not scaled it will grow faster, if there are more misses, which will enable the home bank application to protect its allocation. The pain of losing $painWays$ for the application running on tile j is calculated using the following formula:

$$Pain_{\text{j,painWays}} = \frac{a_{painWays}}{m} \quad (2)$$

where a is the number of misses that will be incurred if the allocation is decreased with $painWays$ and m is the MLP. Our evaluation in Section IV shows that the pain/gain heuristics leads to good cache allocation decisions. We leave further optimization of the pain and gain measures used in this study for future work.

3) Monitoring hardware: We adopt Qureshi's UMON sampled tag array [1] in this work. The original UMON mechanism can predict the number of cache misses under all possible cache allocations (at a single way granularity) based on the access pattern the application has exhibited before. The coarse-grained UMONs, that we use, work by tracking the number of accesses to a shadow tag at coarse-granularity (corresponding to 4 ways). The number of tags required will still be the same but the associated way-hit counter overheads are reduced. The solution also uses dynamic set sampling to decrease the overhead of the monitoring hardware, like the original proposal.

4) Hardware-based implementation: The inter-bank and intra-bank algorithms are implemented in hardware owing to its low computational complexity (see Section IV-E for details). To implement the algorithms each LLC bank controller is provisioned with an ALU capable of computing the pain and the gain and for comparing the values. The inter-bank scheme requires, for each bank, a register array with $N{+}2$ entries, with $log_2(N)$ bits per entry, to store the pain values of other banks. In addition, each bank also includes a register array with $N{+}1$ entries, with $log_2(N)$ bits per entry, to store the id of other tiles in increasing order of distance to determine the next tile to send the challenges to. The intra-bank algorithm leverages the state used by the inter-bank algorithm for establishing allocations.

C. DELTA Enforcement

DELTA's enforcement mechanism has two components. The inter-bank enforcement mechanism utilizes a CBT (the detailed design is presented later in this section) which contains the mapping between address ranges and cache banks. The CBT permits the allocations to span multiple banks by mapping portions of the address space to different banks. The CBT is accessed in parallel with the L2 cache to determine in which LLC bank a certain address is mapped to. The intra-bank enforcement mechanism comprises a standard way-partitioning (WP) unit that keeps track of which ways in the cache bank each core is allowed to insert lines into.

1) Bank partitioning: Each core has a CBT which is organized as a small fully-associative range table that holds the information of address range to bank translation. The CBT works with physical block addresses. We use a simplified version of the range based organization proposed by Gandhi et al. [20]. The total amount of storage required for each CBT is $log_2(N) \times N$ bits, where N is the maximum number of distinct ranges, which is equal to the number of cores/banks. The number of used entries (i.e. ranges) in the CBT is equal to the number of LLC banks allocated by the local core. Only in the rare scenario in which a a single core is active, can a core's CBT grow to map the majority of the chip resources. Therefore, in practice this value is much smaller than the total number of banks and the cost of the associative look-up is negligible.

The CBT is updated when the allocated capacity for a tile expands to/retreats from another tile. When the CBT is updated there is a remapping of address ranges to cache banks. The size of the address range mapped to a bank is proportional to the size of the allocation in that cache bank. Examples illustrating when and how the CBT is updated as allocations expand/retreat are presented in Section II-D.

There are two important design choices for the CBT: (i) how many bits to use, and (ii) which bits to use, i.e. how to map addresses to cache banks. We evaluated different options and found that by using just 8 bits following the set index, as shown in Figure 2, it is possible to effectively distribute the footprint of the application across the different banks. We reverse the bits before we index into the CBT to obtain the bank mapping. The reversing operation turns the least significant bits with the highest entropy to the most significant, which proves to be a reasonable solution for mapping addresses uniformly for the applications we consider.

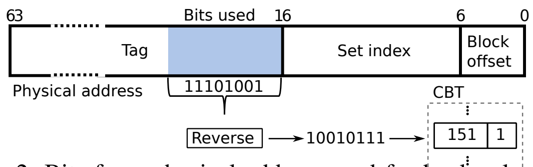

Fig. 2: Bits from physical address used for bank selection.

2) Way partitioning: During insertion, the WP unit uses a bitmask to indicate which cores can insert into a given way in a cache bank. All cores can however access data irrespective of which way it resides. Way-partitioning enforced using bitmasks is practical and has been implemented in commodity systems [19]. The total amount of storage required for each WP unit is $N \times W$ bits, where N is the number of cores and W is the number of LLC ways. DELTA can also work with other fine-grained intra-bank partitioning schemes proposed in literature [14], [15], [21].

3) Invalidation support: A common strategy to handle change in the mapping of an address to a LLC location is to invalidate the line in the cache bank where it currently resides. This invalidation is done by flushing the cache line. Several commodity systems provide ISA level support for this [22]. This is widely used by page coloring mechanisms [23]. However when remapping a large range there is increased overhead due to additional instructions needed for invalidating each address. We therefore rely on hardware support for performing bulk invalidation efficiently. The bulk invalidation unit works by checking the tags to identify addresses that fall in the specified range and invalidates them. This approach does not incur the instruction overhead of cache flushes.

D. Putting it all together

We clarify how the partitioning solution works with the help of two examples that illustrate the different use cases.

Example 1, Inter-bank expansion. The capacity allocation expands to a different tile when a challenge is successful. In Figure 3 we show the process of expansion into a new tile, as well as the state of the CBT and WP before/after the change. We assume that tile 4 has capacity allocated in cache banks in tile 4 and 0, as indicated in the CBT for tile 4. Since, the core in tile 4 sees a considerable gain from expanding its allocation it issues a challenge to tile 5 (#1). Tile 5 compares the gain coming from tile 4 with its own pain of losing cache capacity (#2). Since the gain for tile 4 is considerably larger, tile 5 decides to assign $interDeltaWays$ of ways from its allocation to tile 4 and updates its WP unit (#3). In this case, ways 12-15 are assigned to tile 4 and a response message is sent to tile 4. On receiving the message the tile updates its CBT to also include tile 5 (#4). This is followed by remapping addresses in range 192-255 from tile 4 to tile 5, and invalidating them where they were previously located (#5). Note that expansion process does not require invalidations in tile 5.

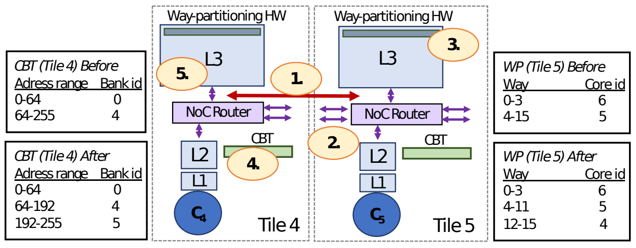

Fig. 3: Example of expansion.

Example 2, Intra-bank algorithm and retreat. The intra-bank algorithm determines whether allocations within a bank expand or shrink. The decision on which partition expands or shrinks is based on the gain of the different applications that share the cache bank. Whenever a partition expands or shrinks the WP unit is updated to reflect the new allocation for the partitions. A shrink will result in a retreat if a partition loses all the ways it was assigned in the cache bank. This scenario is shown in Figure 4. After the intra-bank algorithm is triggered in tile 5 the algorithm decides that tile 4 must give up the entire capacity in the cache bank since it has the least gain among

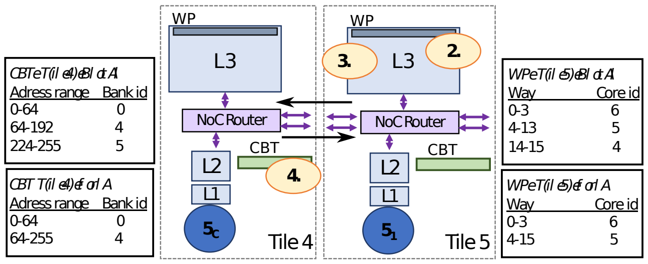

Fig. 4: Example of retreat.

those sharing the cache bank (#1). As a consequence, the WP unit is changed to show the new configuration where ways 14-15 are reassigned to tile 5. The change triggers a message back to tile 4 informing about the retreat. The CBT in tile 4 will now be updated (#2). The addresses previously mapped to tile 5 are remapped, in this example addresses in range 224 to 255 are now remapped back to tile 4. All addresses in the range will consequently be invalidated in tile 5 (#3).

The detailed evaluation of DELTA is presented in Section IV. We show in Section IV-E that the overheads introduced by DELTA are marginal.

E. Support for multithreading

When running multi-threaded workloads containing shared data, accesses to the same line from two different tiles will end up inserting the blocks in two different LLC banks, breaking coherence. To address this, we propose to distinguish between private and shared data at a page granularity and handle them differently. Detection of shared pages including cross-process sharing is performed by the TLB. We adopt a classification scheme, proposed in R-NUCA [24] and used in other NUCA schemes [4], to dynamically classify pages as private or as shared incrementally and lazily. Lines belonging to shared pages are mapped to the cache banks using a fixed S-NUCA strategy whereas lines belonging to private pages are mapped to banks based on the mappings available in the CBT. When a page is first classified as shared all the lines belonging to the page are invalidated.

The allocation algorithm also needs a minor change. On receiving a challenge, the processIDs of the different threads are compared, and the challenge will only be successful if they are different. The rationale is to not let threads from the same application (homogeneous multi-threaded) compete for capacity since it can adversely impact application progress. We expect the performance of this extension with multi-threaded application to be similar to R-NUCA since private data and most of shared data are dealt with in a similar way. Multi-threaded workloads are analyzed in Section IV-C.

III. Experimental Methodology

A. Simulated Architecture

We evaluate our proposal on a 16/64 core tiled CMP architecture modeled using the Sniper Simulator [25]. Details about the baseline architecture are shown in Table II. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and a LLC bank of 512KB. The cache latencies assumed have been modelled using CACTI 6.5 [26].

Cores	16 / 64 cores, x86-64 ISA, 4GHz, OOO, Nehalem-like, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, inclusive, 9-cycle data and 2-cycle tag latency, LRU
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 / 8x8 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 / 8 MCUs, 1 channel/MCU, latency 80 ns, 12.6GB/s per channel
DELTA parameters	reconfiguration interval i_{inter}=1ms i_{intra}=0.1ms, $gainThreshold$=0.5, $minWays$=4, $interDeltaWays$=4, $intraDeltaWays$=1, $gainWays$=4, $painWays$=4

TABLE II: Configuration of the simulated 16- and 64-core tiled CMP.

In Section IV-E we demonstrate that state-of-the-art allocation algorithms, Lookahead or Peekahead, cannot compute locality-aware allocations in a scalable manner. This is because the time needed to compute allocations and locality-aware placement far exceeds the 1 ms reconfiguration interval that we target in this study especially as we scale to larger core counts. In order to fairly compare our distributed solution against the centralized solutions, we model an *ideal centralized* solution that calculates both allocations and locality-aware placement in zero time (no overhead). The ideal solution represents an upper bound on dynamic allocation decisions using the best known centralized algorithm, Lookahead. We use Lookahead as a reference since Peekahead too computes the same allocations as Lookahead albeit with lower overhead. The ideal centralized scheme uses the DELTA enforcement mechanism to support locality-aware mapping in banked LLCs. UMONs are used in each core to measure misses for all possible cache capacity allocations for each application. The cost of invalidations that occur due to remapping of addresses to banks (invalidation+re-fetch) are modelled in detail for both DELTA and the ideal centralized scheme. In addition, we also evaluate an unpartitioned, static NUCA implementation with line-interleaved LLC addresses (unpartitioned S-NUCA), and private LLC, with equal static partitioning of capacity per core (private) for comparison.

DELTA dynamically considers allocations in increments of 32KB from a cache size of 128KB up to 6MB (per application) for the 16-core configuration and 128KB to 24MB for the 64-core case. Each core reserves a minimum of 128KB (see Table II) in the LLC to avoid potential back-invalidations due to the inclusive cache hierarchy.

B. Workloads

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [27]. Workload mixes are constructed by classifying applications in one of the four categories - *cache-insensitive*, *cache-sensitive low*, *cache-sensitive low medium* and *thrashing*, depending on sensitivity to different cache sizes. The classification is performed by running each application for 1B instructions (after fast-forwarding for 1B instructions) with cache sizes of 128KB, 512KB and 8MB. The applications that show improvement in IPC of over 10%, as the cache size increases, are classified as sensitive for a particular cache-size region. Sensitive applications that show improvement in the 128KB to 512KB region are classified as cache-sensitive low and those that also show improvement in the 512KB to 8MB are classified as cache-sensitive low medium. The detailed classification is presented in Table III. Lastly, cache insensitive and thrashing applications experience less than 10% improvement in the 128KB to 8MB range. Among these, we classify applications with a number of Misses-Per-Kilo-Instruction (MPKI) above five as thrashing and the rest as cache insensitive.

Insensitive (I)	povray(po),sjeng(sj),namd(na), zeusmp(ze),GemsFDTD(Ge)
Thrashing (T)	bwaves(bw),libquantum(li),milc(mi)
Cache-sensitive low (L)	h264ref(h2),gromacs(gr),astar(as), gamess(ga),lbm(lb),tonto(to), wrf(wr),leslie3d(le),hmmer(hm)
Cache-sensitive low medium (LM)	dealII(de),omnetpp(om),xalancbmk(xa), gobmk(go),bzip2(bz),gcc(gc),mcf(mc), soplex(so),perlbench(pe),sphinx3(sp), calculix(ca),cactusADM(cac)

TABLE III: Classification of SPEC CPU2006 benchmarks.

Name	Composition	Benchmarks
w1	LM	de,om(2),pe,ca,bz,go(2),ca,hm,le,go,bz,gc,so,mc
w2	L+LM	bw,sj,na,ze,li,mi,ca,sp,de,om,go,go,bz,gc,mc,pe
w3	T+L	to(2),bw(3),lb(2),li(3),h2,mi,gr,as,ga,mi
w4	T+LM	delII,bw(3),so,li(2),hm,pe,mi(3),go,om,bz,go
w5	I+L+LM	gc,po,Ge,as,pe,wr,ga,cac,to,hm,sj,h2,bz,ze,gr,so
w6	I+T+L+LM	na,de,li,gr,wr,so,mi,as,mi,to,ze,om,bw,h2,Ge,hm
w7	I+T+LM	sj,bw(2),bz,wr,li(2),gc,mi,de,na,om,ze,mi,go,Ge
w8	I+T+L	po,bw(2),h2,sj,li(2),gr,na,mi,as,Ge,ga,wr,lb,mi
w9	I+LM	po,om,sj(2),go,na(2),le,ze,go,Ge,bz,wr,ca,sp,gc
w10	I+L	po,to,sj,h2(2),na,lb(2),ze(2),gr,Ge,as,wr,ga,po
w11	T+L+LM	sp,bw,h2,om,li,gr,go,mi(2),as,hm,bw,ga,le,lb,calulix
w12	random	go,lb,ca,sp,bw,go,li(2),ga,h2,ze,to,so,gr,mi,pe
w13	random	lb,to,pe,go,gc,mi,li(2),na,h2,cac,ze(2),ca,so,as
w14	random	de,bw,mc,li,pe,mi,ca,wr,go,po,hm,na,go,ze,so,Ge
w15	random	to(2),po,lb,li,mi,lb,wr,h2,sj,gr,na,as,ze,ga,Ge

TABLE IV: Workload mixes.

We construct a total of 15 workload mixes by combining the applications from the categories described above. Applications from each category are picked randomly while not allowing duplicates unless all applications in a category have already been picked. Details about workload mixes are presented in Table IV. We construct workload mixes for 64 cores by replicating the 16-core workload four times. The applications in a workload mix are mapped to cores randomly.

C. Methodology

We fast-forward for 8B/2B instructions for the 16/64 core simulations. Detailed simulations are carried out until all benchmarks have completed at least 500M/125M instructions and statistics are reported based on the first 500M/125M instructions for each application. We simulate fewer instruction in fast-forward and detailed mode for 64-core CMP to reduce simulation time. The methodology is in line with earlier works [4], [14], [15].

D. Metrics

We use IPC as a measure of performance. We report the geometric mean of IPCs of the applications in a workload, as a performance metric for the workload. We also report the following fairness and throughput metrics: average normalized turnaround time (ANTT) and system throughput (STP) [28]. ANTT is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i}{CPI_{i,private}}$ and STP is given by $\sum_{i=1}^{N}\frac{CPI_{i,private}}{CPI_i}$ ANTT and STP are commonly used for performance evaluation of multi-programmed workloads.

IV. Evaluation

We first compare DELTA against alternative cache organizations and allocation algorithms (described in Section III-A). Next, we show results for multithreaded applications followed by the impact of reconfiguration frequency. Finally, we provide an analysis of DELTA's overheads.

A. Multi-programmed mixes on 16-core CMP

Figure 5 shows the performance for multi-programmed mixes normalized to the unpartitioned S-NUCA. On average, DELTA improves performance by 9% (up to 16%) over S-NUCA whereas the ideal centralized solution shows an average improvement of 12% (up to 22%). In comparison, the private scheme shows an average improvement of 3% over S-NUCA. The results for comparing the fairness and throughput of DELTA and the ideal centralized scheme are shown in Figure 6. On average, DELTA is 2% behind in terms of ANTT and 5% behind in terms of STP, than the ideal centralized scheme. Note that a lower value signifies greater fairness with ANTT while a higher value is equivalent to larger throughput with STP.

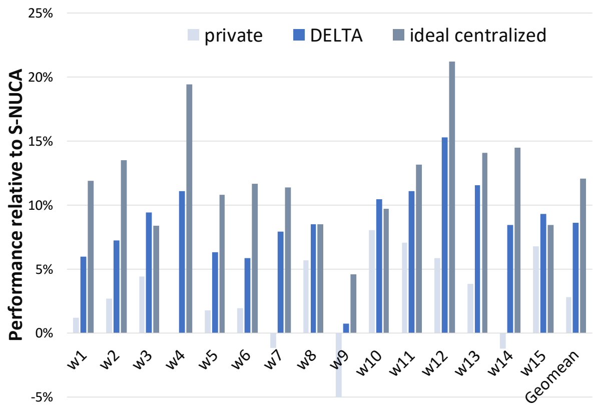

Fig. 5: Performance of workload mixes normalized to unpartitioned S-NUCA on a 16-core CMP.

As can be seen in Figure 5, the ideal centralized scheme is better than DELTA in 11 out of 15 mixes. In four cases DELTA performs on par or better. In order to understand the performance gap between the ideal centralized scheme and DELTA we investigate a single workload in detail. Figure 7 shows the performance of different applications in a single

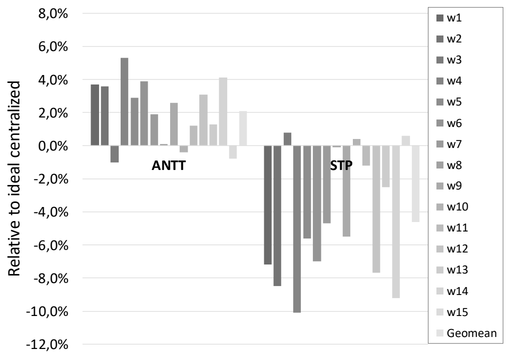

Fig. 6: Fairness comparison between ideal centralized and DELTA.

workload using the ideal centralized scheme, normalized to DELTA (in gray). In addition, it also shows the performance of the private scheme normalized to DELTA (in blue). As can be seen in the figure, most of the applications perform almost identically with the exception of xalancbmk and soplex. For these two applications, the ideal centralized solution performs considerably better than DELTA, by 45% and 35%, respectively. Note that DELTA performs better than the private scheme for these two applications by 12% and 36%.

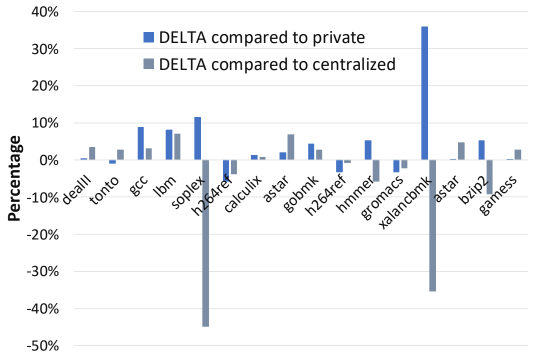

Fig. 7: Normalized performance for applications in *w2* on a 16-core CMP.

To understand the trend for xalancbmk and soplex, we compare the allocations of cache capacity, in terms of number of ways, made by the ideal centralized scheme and DELTA, for the different applications in the workload. The ideal centralized algorithm gives a larger allocation of 42 respectively 50 ways on average to xalancbmk and soplex in comparison to DELTA which gives 26 and 20 ways. This behaviour can be attributed to the *farsighted* nature of the ideal centralized scheme i.e. it uses information about the entire miss curves for all applications to determine allocations. DELTA, in contrast, is *nearsighted*, i.e. uses a limited window of the miss-rate curves to determine the pain/gain which influences allocations. As xalancbmk and soplex do not see a considerable improvement in the limited window, DELTA does not allocate as much cache capacity as the ideal centralized scheme. The difference in the size of allocations impacts the performance because these applications are sensitive to additional cache capacity.

In Figure 8 we show the performance of individual applications in one of the workloads where DELTA is on par with the ideal centralized scheme. We see that individual applications mostly perform as well as or better than the centralized scheme even though DELTA is nearsighted. The same trend holds also for the other workloads (w3,w8,w10,w15).

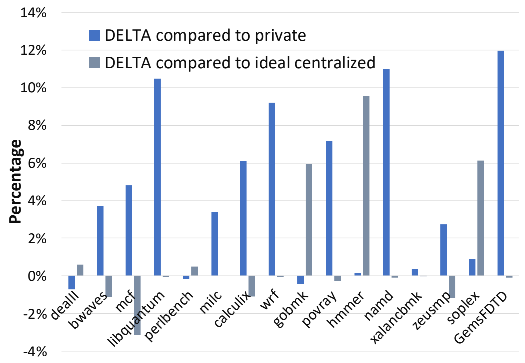

Fig. 8: Normalized performance for applications in *w3* on a 16-core CMP.

B. Multi-programmed mixes on 64-core CMP

To investigate how our proposal scales to larger core counts we evaluate a 64-core CMP. Figure 9 shows the performance for the individual multi-programmed workload mixes. DELTA, on average, improves performance by 16% (up to 28%) over S-NUCA, while the ideal centralized scheme improves performance by 17% (up to 35%). The private scheme performs better for 64-cores than for 16-cores, but is generally regarded as an inefficient solution since it cannot handle underutilized scenarios. The results for comparing fairness and throughput between ideal centralized and DELTA indicate that the difference between the two schemes is 1% for STP and less than 1% for ANTT (not shown). The results also indicate that DELTA makes good allocation decisions, on par with an ideal scheme, in spite of the distributed nature of the algorithm that increases the number of re-configurations (steps) required to span across all the banks in a CMP.

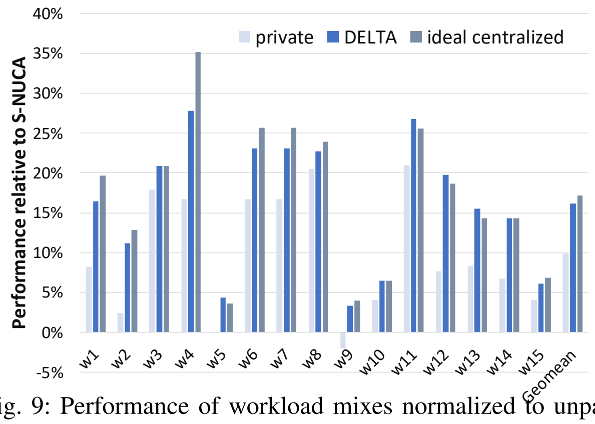

Fig. 9: Performance of workload mixes normalized to unpartitioned S-NUCA on a 64-core CMP.

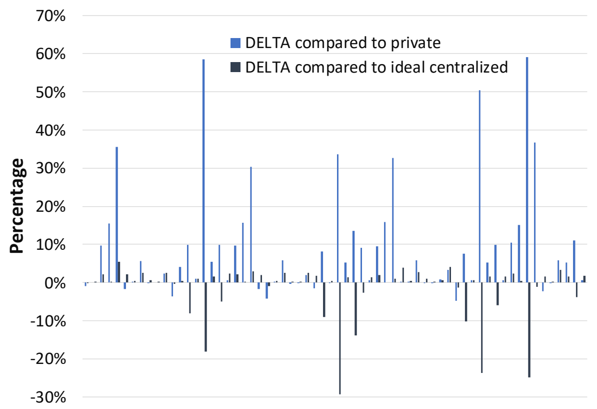

Fig. 10: Normalized performance for applications in *w2* on a 64-core CMP.

The performance gap between DELTA and the ideal centralized scheme has diminished on average, compared to the results for a 16-core CMP. For seven workloads (w3, w5, w10, w11, w12, w13, w14) DELTA is on par or better than ideal centralized. For workload w2, as shown in Figure 10, we observe that DELTA falls behind the ideal centralized scheme in the 64-core CMP experiments, similar to the trend seen with the 16-core CMP. For many applications in this workload DELTA is still better than the ideal centralized scheme but in the few cases where the reverse is true, the difference in performance is comparatively larger. For applications such as xalancbmk and soplex the ideal centralized scheme surpasses DELTA by giving a larger allocation, because it is farsighted.

We investigate one of the workloads (w13) where DELTA performs better than the ideal centralized scheme, as shown in Figure 11. The farsightedness of the ideal centralized scheme results in it allocating over 250 ways to applications such as lbm and libquantum. Moreover, the centralized scheme does not consistently detect the benefit of giving these applications a large allocation, and switches the cache capacity allocation between a large and small allocation. In general, giving larger allocation to a few applications puts severe constrains on the allocations for the other applications and degrades the overall performance. DELTA does not suffer from making these unadvantageous allocations and performs better for the mixes containing applications like lbm and libquantum.

In summary, the evaluation shows that DELTA performs almost as well as the ideal centralized scheme as we scale to 64 cores. Furthermore, this demonstrates that a dynamic distributed scheme can give good allocations, without incurring the overheads associated with computing allocations centrally.

C. Multi-threaded applications

We estimate the performance of DELTA using SPLASH2 suite in order to understand how the scheme performs with multithreaded applications. Figure 12 shows the speed-up obtained by DELTA over the S-NUCA implementation and compares it to the private cache configuration. We execute each application on the 16-core CMP and using large input sets (from Sniper) to obtain performance data for the baselines. We use number of cycles for the longest running thread within

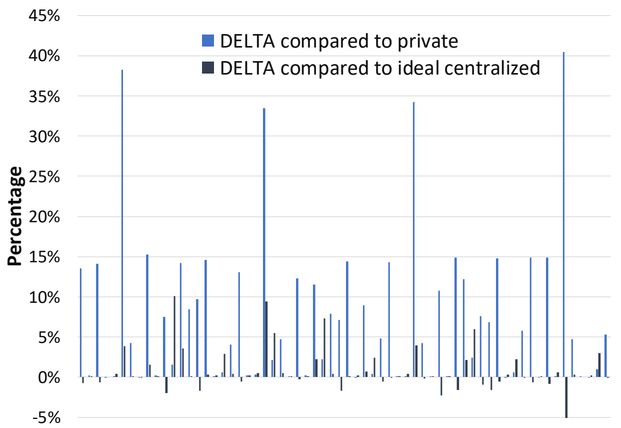

Fig. 11: Normalized performance for applications in *w13* on a 64-core CMP.

the parallel region, which we identify as the region of interest (ROI), as a measure of performance.

We follow a two step process to estimate the performance of DELTA. Firstly, we measure the ratio of private/shared pages and cache blocks. These results are shown in Table V. For this we develop a pintool [29] that instruments all loads and stores in the region of interest to measure inter-thread sharing at page and cache block granularity. Next, we estimate the performance of DELTA by performing a piece-wise reconstruction of the execution in which private accesses are modeled according to private LLC baseline's performance, and shared accesses are modeled according to the S-NUCA baseline performance. To simplify, we assume that the LLC accesses are uniformly distributed across pages. Private pages are reclassified at most once, and the S-NUCA mapping is never reverted. Hence, for long running applications this overhead is negligible. We expect the estimation to be accurate since DELTA maps lines from private-pages to the private bank and utilizes S-NUCA mapping for lines from the shared pages.

By design (see Section II-E), the performance of DELTA is usually between the performance of the S-NUCA and private baselines, depending on the amount of private/shared pages. Over the entire SPLASH2 suite, the average performance of DELTA compared to both the private LLC configuration and S-NUCA configuration is within 1% for both cases. The actual

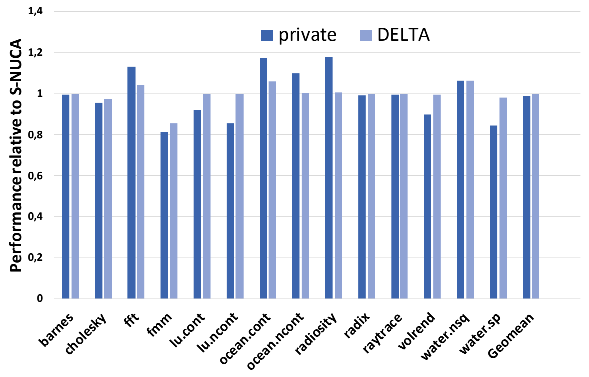

Fig. 12: Normalized performance for splash2 on a 16-core CMP.

App.	barnes	cholesky	fft	fmm	lu.cont	lu.ncont	ocean.cont
Page	8.2	62	33	73	0.5	0.5	38
Block	9.3	66	34	65	0.3	0.3	98.6
App.	water.sp	radiosity	radix	raytrace	volrend	water.nsq	ocean.ncont
Page	10	3	5.2	17	5.7	99.8	1.1
Block	70	4.6	10	24	21	91	99

TABLE V: Percentage of private pages and blocks.

performance for each benchmark depends considerably on the amount of sharing, with variations of up to 20%. For example, in `lu.ncont`, which has high ratio of sharing (>99%), DELTA's performance is almost equal to the S-NUCA performance, while the private LLC configuration suffers performance loss of approx. 10%. On the other hand, in `water.nsq`, where almost all pages are private, DELTA's performance is equivalent to that of the private LLC configuration, achieving a speed-up of 6% over S-NUCA.

The results in Table V indicate that several benchmarks have a low amount of private pages as opposed to private blocks. On reason for this is the existence of shared data, such as boundary elements in structured grid simulations, in pages that are mostly private. In general, this will lead to less locality-optimized cache access for these benchmarks. Due to the additional costs associated with distant memory accesses (both in DRAM and in caches), modern HPC software development encourages programming styles that result in higher number of private pages. This is designed to ensure threads operate on local memory thereby reducing the amount of shared pages [30]. Moreover, an important trend in algorithm design are Communication-avoiding Algorithms (CAA) [31] which, attempt to reduce sharing. Hence, we expect the architecture of DELTA to achieve even better performance on modern multi-threaded workloads with a considerable amount of private data in comparison to S-NUCA. Detailed modeling and optimization of DELTA for emerging multithreaded workloads is left for future research.

D. Frequency of reconfiguration

In order to understand the impact of the frequency by which cache allocations are computed, we simulate a cache partitioning solution that uses an ideal implementation of Lookahead for computing allocations with zero overheads (see Section III-A for details about the ideal centralized implementation) at two cache allocation frequencies (1 ms and 100 ms), on the baseline system. Figure 13 shows the impact of allocation frequency on performance of five different workload mixes each comprising 16 SPEC CPU2006 benchmarks (see Table IV). The results demonstrate that while frequent allocations do not benefit all workloads, they do provide the opportunity to improve performance for several of the workloads considered, because of better adaptation to phase changes.

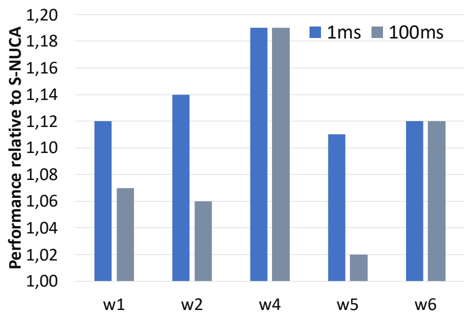

Fig. 13: Impact of frequency of reconfigurations on a 16-core CMP.

E. Overheads

We analyze the different sources of overheads in the centralized scheme and DELTA.

1) Computational overheads: The worst-case time complexity of the Lookahead algorithm is $O(N \times W^2)$ where N is number of cores and W is number of ways. We can consider the algorithm to have cubic complexity, since the number of ways needs to be at least as many as the number of cores (for way-partitioning). The best case complexity is $O(N \times W)$, i.e. quadratic. Peekahead, which considers only the points of the miss rate on a convex hull, has a complexity of $O(N \times W)$, in the best/average case and $O(N \times W^2)$ in the worst-case. The time to compute cache allocations for different core counts using Lookahead and Peekahead, with 16 ways per core, is presented in Table VI.

The Lookahead algorithm takes 5.32 ms on average to compute allocations for a CMP with 16 cores (16-tile CMP with each bank containing 16 ways). Peekahead takes 0.89 ms on average for the same scenario. For larger core counts the overhead is even larger. Note that the data presented in the table do not take into account the additional computations needed to perform locality-aware data placement, which for large core counts has been shown to exceed capacity allocation overheads [32].

For DELTA the complexity can be attributed to the inter- and intra-bank allocation algorithm. The pain and gain computation step takes constant time, i.e. $O(1)$ complexity. The inter/intra bank allocation algorithm requires finding the core with the MIN and MAX gain/pain values. This operation is similar to finding the min. and max. in an unsorted array. The simplicity of the DELTA reconfiguration algorithms enables a hardware implementation with low overheads. Even if the algorithms were to be implemented in software, the overhead of DELTA's inter- and intra-bank allocation algorithms assuming a 64-core CMP would be 0.015 ms and 0.007 ms, three orders of magnitude lower than state-of-the-art.

2) Message overheads: We calculate the number of additional messages sent in the worst-case at each reconfiguration interval (assuming $i_{inter} = 1ms$ and $i_{intra} = 0.1ms$) in a 16-core CMP. For the centralized scheme the total number of messages is $2 \times N$, where N is the number of cores in the system, and this results in $16 \times 2 = 32$ additional

Cores	**2**	**4**	**8**	**16**	**32**	**64**
Lookahead	0.02	0.05	0.46	5.32	73.07	1230
Peekahead	0.03	0.07	0.23	0.89	3.34	13.12

TABLE VI: Overhead for Lookahead and Peekahead in ms per invocation.

messages. DELTA however needs $2 \times N$ messages for intra-bank allocation and $N \times 10 \times 2$ for inter-bank allocation which results in a total of 352 messages. However, the number of messages on the NoC pertaining to L2 misses alone, during the same interval, is 320K on average. This indicates that the overhead in terms of additional messages for DELTA is marginal ($\sim$0.1%), even in the worst-case.

3) Invalidation overheads: Invalidations are needed when remapping addresses to LLC locations regardless of whether the allocation algorithm is centralized or distributed. Invalidation overheads can be primarily attributed to two causes: the overheads of performing the invalidations and how often/much data need to be invalidated. We address the first by performing bulk invalidations (see Section II-C). We mitigate the second by only requiring invalidations for inter-bank reconfigurations. Intra-bank reconfigurations do not lead to invalidations except when a retreat is triggered in rare cases (see Section II-D).

V. Related work

Cache Partitioning: Several solutions have been proposed in literature for cache resource partitioning. Way partitioning is the most popular and it is implemented in commodity systems [2], [19], [33], [34]. It is a simple technique that works by limiting which ways a core can insert into. The major limitation is that it requires cache associativity to scale with the number of cores, which is not easily done [35]. Set partitioning is another approach, which can be implemented with hardware or software support [8], [9], [23], [36]. Hardware based schemes require flexible indexing of the cache. Software schemes use page coloring and rely on OS support for partitioning. Page coloring, however, cannot support superpages and incurs high overhead for reconfiguration. The aforementioned solutions also have the drawback of only supporting a limited number of coarse-grained partitions.

Fine-grained partitioning solutions can be broadly classified in three categories, i) hybrid techniques [10], ii) clustering techniques [11], [12] and iii) replacement-based techniques [13]–[16]. Hybrid techniques like SWAP, combine set and way partitioning in order to get more fine-grained partitions. Clustering techniques like KPart, group applications into clusters and then assigns clusters to way partitions, to emulate fine-grained partitioning. The replacement-based techniques adapt the cache replacement policy to enable fine-grained partitions with different sizes. However, in the context of tile-based CMPs, these approaches leave room for further improvement since they do not take locality into account.

A few proposals performs locality-aware placement for tiled CMPs [4], [18]. CloudCache [18] uses virtual private cache partitions that span across banks and performs locality-aware placement of the partitions. The drawback of this proposal is that it uses N-chance spilling [37] on evictions and requires costly broadcasts. Jigsaw [4] lets software define *shares* and then maps data to them by assigning a share *id* to every page in the application. Allocation and enforcement is done at share granularity instead of application/core granularity as in prior proposals. The proposal relies heavily on software support. Furthermore, in the aforementioned solutions the allocation decision is made by a central hardware or software component which limits scalability.

To lower the overhead of a central component, XChange [38] uses a market-based approach where some of the computations for multi-resource management are done in each core/bank. However when used for partitioning a single resource, XChange will result in an equal partitioning. Moreover the scheme is based on a shared L2 cache structure and thus does not enable locality-aware placement, unlike DELTA.

Non-Uniform Cache Access (NUCA): Efficient usage of NUCA caches has been an extensively researched topic [24], [39]–[45]. The simplest approach, Static NUCA (S-NUCA), spreads the data over all cache banks with a fixed mapping and exposes variable access latencies.

D-NUCA schemes try to combine the best from private and shared cache designs, where private designs have isolation but suffers from under-utilization and shared designs have dynamic utilization but suffers from long on-chip latencies and interference.

A few proposals, like DELTA, try to minimize long on-chip distance. They mostly focus on multi-threaded applications and achieve this by replication and placement, where frequently used lines are copied and placed in the nearest cache bank [42]–[44]. Spilling has been proposed as a way to overcome the problem of underutilized private caches. It works by inserting a copy of a line in another cache bank before it is evicting from the cache hierarchy [37], [46].

Both spilling and replication have two problems. Firstly, both operations decrease the capacity of the cache, where a tradeoff is made between latency and capacity. Secondly, costly directory lookups are needed in order to find data. In our proposal, we avoid both these drawbacks since we place data in a locality-aware way without replicating and do not need a directory to find the data.

Some D-NUCA proposals implement a different partitioning strategy, where separation is done between shared and private regions instead of applications [47]–[50]. Elastic Cooperative Caching (ECC) [50] uses a distributed approach to divide the cache bank between shared/private using way partitioning. The scheme also uses spilling to extend capacity to other banks and therefore inherits the drawbacks of spilling. In contrast to this work we enforce strict per application partitions that can span multiple banks.

R-NUCA [24] classifies accesses into three categories (instructions, private data, shared data) and uses static rules for placement and replication for each type. The drawbacks of the scheme is that it uses a static placement scheme for the different classes not taking dynamic nor application specific behaviour into account, which DELTA does. Note that, compared to DELTA, none of the aforementioned techniques give strict interference protection for data.

Coherence framework: CDR [51] reduces the scope of cache coherence from global to VM-, application-, or page-level to enable shared memory between servers or to minimize on-chip distances in a manycore. Unlike DELTA it does not provide strict cache partitioning. Furthermore, creation of sharer domains does not take the cache requirements of each application into account but instead allows the different threads of the same application to form a domain.

To the best of our knowledge, DELTA is the first distributed solution for fine-grained and locality-aware cache partitioning which permits permits hardware implementation and scales to many-core architectures.

VI. Conclusions

We present DELTA, a fully distributed and locality-aware partitioning solution for tile-based CMPs. The solution is scalable through its novel challenge-based allocation algorithm, which allocates cache capacity in a distributed way based on the performance gain of each application.

We show that the distributed algorithm has low computational overhead which permits hardware implementation and enables frequent reconfiguration. The allocation algorithm is supported by a flexible enforcement mechanism that enables locality-aware placement. Our evaluation demonstrates that the distributed partitioning solution performs close to an ideal centralized solution and scales to large core counts.

Acknowledgment

This research has been funded by the European Research Council, under the MECCA project, contract number ERC-2013-AdG 340328 and the Swedish Research Council (VR), under the ACE project. The simulations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

References

[1] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.
[2] L. R. Hsu *et al.*, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *PACT-15*, 2006.
[3] D. Thiebaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. on Comp.*, 1992.
[4] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.
[5] R. Riesen *et al.*, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, 2009.
[6] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proc. SC'10*, 2010.
[7] T. Jones, "Linux kernel co-scheduling for bulk synchronous parallel applications," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 57–64.
[8] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. HPCA-14*, 2008.
[9] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. MICRO-39*, 2006.
[10] X. Wang *et al.*, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *Proc. HPCA-23*, 2017.
[11] H. Cook *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ISCA-40*, 2013.
[12] N. El-Sayed *et al.*, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. HPCA-24*, 2018.
[13] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, 2013.
[14] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management," in *Proc. ISCA-39*, 2012.
[15] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.
[16] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.
[17] R. Iyer, "Cqos: A framework for enabling qos in shared caches of cmp platforms," in *Proc. ICS-18*, 2004.
[18] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.
[19] A. Herdrich *et al.*, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proc. HPCA-22*, 2016.
[20] J. Gandhi *et al.*, "Range translations for fast virtual memory," *IEEE Micro*, 2016.
[21] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. MICRO-47*, 2014.
[22] "Intel® 64 and IA-32 Architectures Developer's Manual."
[23] M. Awasthi *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. HPCA-15*, 2009.
[24] N. Hardavellas *et al.*, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proc. ISCA-36 2009*, 2009.
[25] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.
[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO-40*, 2007.
[27] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002.
[28] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[29] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[30] D. Ott, "Optimizing Applications for NUMA," 2011.
[31] J. Demmel, "Communication avoiding algorithms," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 1942–2000.
[32] N. Beckmann, "Design and analysis of spatially-partitioned shared caches," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.
[33] S.-H. Yang *et al.*, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *Proc. HPCA-8*, 2002, pp. 151–161.
[34] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. HPCA-8*, 2002, pp. 117–128.
[35] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proc. MICRO-43*, 2010, pp. 187–198.
[36] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proc. MICRO-34*, 2001.
[37] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proc. HPCA-15*, 2009.
[38] X. Wang and J. F. Martinez, "XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proc. HPCA-21*, 2015.
[39] S. Srikantaiah *et al.*, "MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy," in *Proc. HPCA-17*, 2011.
[40] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The Direct-to-Data (D2D) Cache: Navigating the cache hierarchy with a single lookup," in *Proc. ISCA-42*, 2014.
[41] ——, "A Split Cache Hierarchy for Enabling Data-Oriented Optimizations," in *Proc. HPCA-23*, 2017.
[42] M. Zhang *et al.*, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. ISCA-32*, 2005.
[43] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.
[44] P. A. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A New Approach to Replication in Distributed Shared Caches," in *Proc. PACT-26*, 2017.
[45] Q. Shi *et al.*, "LDAC: Locality-Aware Data Access Control for Large-Scale Multicore Cache Hierarchies," *ACM TACO*, 2016.
[46] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. ISCA-33*, 2006.
[47] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proc. HPCA-13*, 2007.
[48] L. Zhao *et al.*, "Towards hybrid last level caches for chip-multiprocessors," *ACM SIGARCH Comput. Archit. News*, 2008.
[49] J. Merino, V. Puente, and J. A. Gregorio, "ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture," in *Proc. HPCA-6*, 2010.
[50] E. Herrero *et al.*, "Elastic cooperative caching," in *Proc. ISCA-37*, 2010.
[51] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proc. MICRO-48*, 2015.