2017 IEEE International Symposium on High Performance Computer Architecture

Tiny Directory: Efficient Shared Memory in Many-core Systems with Ultra-low-overhead Coherence Tracking

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

***Abstract*—The sparse directory has emerged as a critical component for supporting the shared memory abstraction in multi- and many-core chip-multiprocessors. Recent research efforts have explored ways to reduce the number of entries in the sparse directory. These include tracking coherence of private regions at a coarse grain, not tracking blocks that belong to pages identified as private by the operating system (OS), and not tracking a subset of blocks that are speculated to be private by the hardware. These techniques require support for multi-grain coherence, assistance of OS, or broadcast-based recovery on sharing an untracked block that is wrongly speculated as private. In this paper, we design a robust minimally-sized sparse directory that can offer adequate performance while enjoying the simplicity, scalability, and OS-independence of traditional broadcast-free block-grain coherence. We begin our exploration with a naïve design that does not have a sparse directory and the location/sharers of a block are tracked by borrowing a portion of the block's last-level cache (LLC) data way. Such a design, however, lengthens the critical path from two transactions to three transactions (two hops to three hops) for the blocks that experience frequent shared read accesses. We address this problem by architecting a tiny sparse directory that dynamically identifies and tracks a selected subset of the blocks that experience a large volume of shared accesses. We augment the tiny directory proposal with an option of selectively spilling into the LLC space for tracking the coherence of the critical shared blocks that the tiny directory fails to accommodate. Detailed simulation-based study on a 128-core system with a large set of multi-threaded applications spanning scientific, general-purpose, and commercial computing shows that our coherence tracking proposal operating with $\frac{1}{32}\times$ to $\frac{1}{256}\times$ sparse directories offers performance within a percentage of a traditional $2\times$ sparse directory.**

***Index Terms*—sparse directory; cache coherence; shared memory;**

I. Introduction

Cache coherence protocols are central to the correctness of shared memory abstractions in distributed parallel environments. An important storage structure used by the scalable implementations of such protocols is the coherence directory, which is responsible for keeping track of the current locations of the memory blocks in the cache hierarchy. In a single-chip many-core system, the coherence directory maintains information about the blocks resident in the private cache hierarchy of each processing core. The sparse directory organization [20], [32] has become popular due to its simplicity and space-efficiency. The sparse directory organizes the coherence tracking information in the form of a cache, which can track only a limited number of blocks at a time. For example, in a three-level cache hierarchy with the first two levels being private, if the last level (L2) of the private caches aggregated over all the cores can accommodate N blocks, a $\frac{1}{16}\times$ sparse directory would track at most $N/16$ unique blocks at a time. A replacement from the sparse directory invalidates or retrieves (if dirty) the corresponding block from all the private caches having a copy of the block.

The number of sparse directory entries is an important determinant of end-performance. An undersized sparse directory may experience premature eviction of tracking entries leading to invalidation of the blocks corresponding to the evicted tracking entries. Figure 1 shows the execution time of seventeen multi-threaded applications as the number of entries in the sparse directory is varied in a 128-core system. The results are normalized to the execution time with a $2\times$ sparse directory. All sparse directories are eight-way set-associative.[1] On average, the execution time with $\frac{1}{4}\times$, $\frac{1}{8}\times$, and $\frac{1}{16}\times$ sparse directories increases by 3%, 11% and 28%, respectively, compared to the $2\times$ directory. Ocean_cp is an outlier and improves in performance with decreasing directory size because a smaller directory converts a subset of performance-critical three-hop accesses to two-hop accesses. Overall, reducing the coherence tracking overhead without losing performance is important.

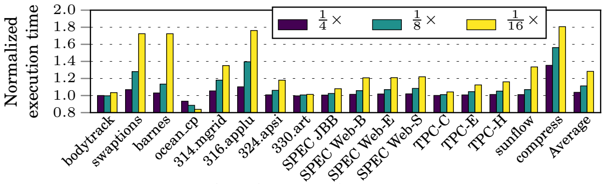

Fig. 1. Performance with $\frac{1}{4}\times$, $\frac{1}{8}\times$, and $\frac{1}{16}\times$ sparse directories normalized to a $2\times$ directory.

A block allocated in the LLC either remains private throughout its residency in the LLC or gets actively shared. To understand the proportion of these two types of blocks, Figure 2 shows the percentage of the allocated LLC blocks that experience a maximum of k distinct sharers during the residency in the LLC where k falls in four possible sharer count bins: 2 to 4, 5 to 8, 9 to 16, and 17 to 128 (end-points inclusive). These data are collected on a 128-core system. The LLC is sized so that the number of blocks is same as the number of entries that a $2\times$ sparse directory would have. These data show that, on average, 21% of the allocated blocks observe sharing, while the rest remain private during their residency in the LLC. While these data do not show the absolute shared footprint, the SPECWeb and TPC

[1] Section II discusses our simulation environment.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.24

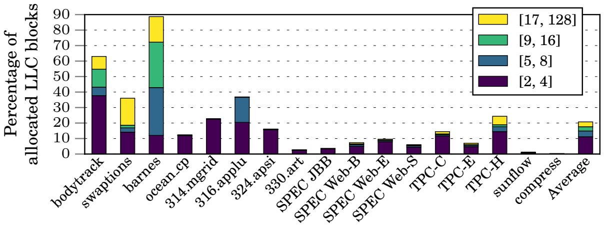

Fig. 2. Distribution of maximum sharer count per allocated LLC block.

benchmark applications have much larger shared footprints than most of the applications and they also carry out a larger number of LLC fills.

Motivated by the observation in Figure 2, recent proposals have explored several ways to reduce the number of sparse directory entries that track private blocks [4], [12], [13], [14], [47]. The data in Figure 2 also indicate that if a sparse directory is dedicated to track only shared blocks, it can be small. We conduct an experiment to find out how small such a sparse directory can be. In this experiment, a block's tracking entry is allocated in the sparse directory only when the block enters the shared state with two distinct sharers. The tracking entry stays in the sparse directory until it is evicted from the directory or the block reaches a state where it has no sharer or owner.[2] As long as a block remains private or is exclusively owned by a core, it is tracked in a special structure of unbounded capacity. It is important to note that if a block exhibits a sharing pattern where it moves from one core to another while staying in an exclusively owned state (E or M in our baseline MESI protocol), it is tracked in the special unbounded structure and does not get allocated in the sparse directory until and unless it enters the S state with two sharers. Figure 3 quantifies the performance of such a design with varying size of the sparse directory dedicated to track only shared blocks. These results completely ignore the overhead of the unbounded special structure that tracks the other blocks. As the size of the sparse directory dedicated to track only shared blocks is set to $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$, and $\frac{1}{128}\times$, the average losses in performance compared to a traditional $2\times$ sparse directory are 1%, 4%, 13%, and 28%, respectively. The $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sparse directories are eight-way set-associative, while the $\frac{1}{128}\times$ sparse directory having just sixteen entries per LLC bank is fully-associative. We have also conducted this experiment with a four-way skew-associative sparse directory that employs a H3 hash-based Z-cache organization [36] for the $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes. In this case, the performance losses are 0.5%, 3%, and 12% respectively for $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes of the sparse directory. These results indicate that even if the entire tracking overhead of non-shared blocks is lifted from the sparse directory, it is not possible to reduce the directory size to $\frac{1}{32}\times$ or less using traditional techniques without suffering from noticeable performance losses.

In this paper, we present a different ground-up approach for designing a robust sparse directory having a minimal number of entries. We retain the simplicity and scalability of a traditional broadcast-free OS-independent block-grain coherence protocol. We begin our exploration with an architecture that does not have a sparse directory and consider the possibility of borrowing a few bits of the LLC data way of the block for tracking coherence information (Section III). In this design, a significant performance problem arises when a block gets shared. Each sharing access received by the LLC must be forwarded to an elected sharer, which can supply the data block to the requester; the LLC cannot supply the correct data block because portion of the LLC data block is corrupted and used to track the sharers. We address this performance shortcoming by architecting a tiny directory, which is a novel sparse directory design for dynamically identifying and tracking a critical subset of the blocks that experience most shared accesses (Section IV). We also introduce the option of selectively spilling a subset of the shared tracking entries into the LLC space when the tiny directory is too small to track the critical shared working subset. This option, however, introduces the new challenge of dynamically deciding the appropriate volume of spills so that the volume of LLC misses does not get affected. Simulation results show that our proposal implemented in a 128-core system operating with a tiny directory of size ranging from $\frac{1}{32}\times$ to $\frac{1}{256}\times$ performs within a percentage of a system with a traditional $2\times$ sparse directory (Section V).

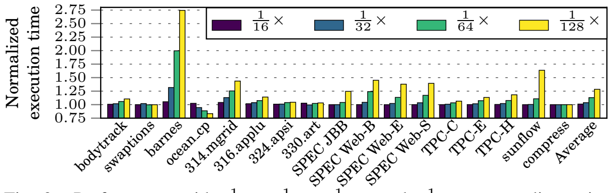

Fig. 3. Performance with $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$, and $\frac{1}{128}\times$ sparse directories for tracking shared blocks only. Tracking non-shared blocks has no overhead. Results are normalized to a $2\times$ sparse directory.

A. Related Work

The early proposals focused on optimizing the coherence directory store in the distributed shared memory multiprocessor architectures. The first proposal on coherence directory design introduced a bitvector as the directory element [6]. Since then several designs have been proposed to optimize the coherence directory storage in the distributed shared memory multiprocessors [1], [2], [3], [7], [8], [9], [11], [19], [20], [23], [28], [30], [31], [42].

More recent proposals have focused on directory space optimization for chip-multiprocessors. Several proposals have attempted to optimize the number of entries in the sparse directory. Smart hash functions and skew-associative organizations for the sparse directory have been proposed [15], [35]. Designs that store the evicted directory entries in a memory-resident hash table and delay invalidations have also been explored [24]. Page-grain classification between private and shared data has been used to exclude private blocks from coherence tracking, thereby effectively increasing the number of available directory entries for tracking shared data [12]. A recently proposed design does not invalidate private blocks on directory eviction, but resorts to broadcast when such a block gets shared after the tracking entry of the block is evicted from the sparse directory [13]. Recent proposals employing coarse-grain coherence tracking for privately cached regions can further reduce the required number of directory entries [4], [14], [47]. Proposals that track a small set of sharing patterns and link each active directory entry to a sharing pattern have been explored [50], [51]. The recently proposed in-cache coherence tracking design uses the entire LLC

[2] In our implementation, all evictions from the private cache hierarchy are notified to the directory [29]. The eviction notices for the blocks in E or S state do not carry any data.

data block of an LLC tag for tracking coherence information for that tag [16]. As we show in Section III, a design similar to this proposal suffers from a large volume of three-hop transactions for shared accesses. Compiler-generated hints about private data have been used to optimize directory allocation [27]. Data-race-free software, disciplined parallel programming models, and self-invalidation of shared data at synchronization boundaries have been used to significantly reduce the coherence directory size or completely eliminate the coherence directory [10], [33], [40].

In this study, we assume each sparse directory entry to be a full-map bitvector and focus squarely on optimizing the number of entries in the sparse directory. However, there have been several proposals that optimize the average number of bits per directory entry [14], [25], [35], [37], [46], [48], [49]. Any standard technique for limiting the width of the directory entry can be seamlessly applied on top of our proposal to further reduce the area of the sparse directory.

II. Simulation Framework

We use an in-house modified version of the Multi2Sim simulator [41] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. The details are presented in Table I. The interconnect switch microarchitecture assumes a four-stage routing pipeline with one cycle per stage at 2 GHz clock. The stages are routing 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. The applications for this study are drawn from various sources and detailed in Table II (ROI refers to the parallel region of interest). 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, SPLASH-2, and OMP 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 pre-processed 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 memory block (64 bytes).

III. In-LLC Coherence Tracking

This section discusses the design of an in-LLC coherence tracking technique which does not have a sparse directory and borrows few bits of the LLC block for tracking coherence information. The design extends a traditional MESI coherence protocol [26]. Section III-A discusses the organization of the coherence states in the LLC. Section III-B introduces the small extensions needed on top of the traditional MESI coherence protocol. We evaluate the in-LLC coherence tracking technique in Section III-C and understand the major shortcomings of this design. This evaluation sets the stage for the tiny directory design, which is our central contribution.

TABLE I
Simulation environment

On-die cache hierarchy, interconnect, and coherence directory
Per-core iL1 and dL1 caches: 32 KB, 8-way, 2 cycles
Per-core unified L2 cache: 128 KB, 8-way, 3 cycles, non-inclusive/non-exclusive, fill on miss, no back-inval. on eviction
Shared L3 cache: 32 MB, 16-way, 128 banks, 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, replacement policy at all levels: 64 bytes, LRU
Interconnect: 2D mesh clocked at 2 GHz, two-cycle link latency (1 ns), four-cycle pipelined routing per switch (2 ns latency); Each hop: a core, its L1 and L2 caches, one L3 cache bank, one sparse directory slice tracking home blocks.
Sparse directory slice: 1-bit NRU replacement, 8-way (fully-associative for $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes)
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 [34], 12-12-12, BL=8, 64-bit channels, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices, open-page policy

A. Organization of Coherence States

A valid LLC block can be in one of three stable coherence states: unowned/non-shared, exclusively owned by a core (in E or M state), and shared by one or more cores. Additionally, a pending/busy state is needed to handle transience. As in the baseline, we assume two state bits per LLC block: valid (V) and dirty (D). These two bits are used to encode four states of an LLC block as depicted in Table III. The state encoding shown in the last row is introduced for the purpose of in-LLC coherence tracking. In this state, the first four bits (denoted b_0, b_1, b_2, b_3) of the data block encode the extended state of the block as shown in Table IV. The number of cores is assumed to be C. In summary, when the LLC block state is (V=0, D=1), either $4 + \lceil \log_2(C) \rceil$ bits or $4 + C$ bits of the data block are corrupted for tracking the extended coherence states.

B. Coherence Protocol Extensions

The in-LLC coherence tracking mechanism minimally extends a traditional baseline write-invalidate MESI coherence protocol. In the baseline protocol, an instruction read access to the LLC is always responded to in S state even if the requester is the only core accessing the block. This helps accelerate code sharing.[4] The baseline protocol assumes that all evictions from the private cache hierarchy are notified to the LLC [29]; the eviction notices for clean blocks do not carry any data. A request that is forwarded to an owner core is responded directly to the requester core with a notification to the home LLC bank for clearing the busy/pending state of the involved cache block. As in the AlphaServer GS320 protocol, a late intervention in the baseline protocol is resolved by the owner core by keeping the evicted block in a buffer until the eviction notice is acknowledged by the home LLC bank [18].

In the in-LLC coherence tracking mechanism, an invalid (V=0, D=0) or unowned (V=1) LLC block enters a corrupted state (V=0, D=1) when it is requested by a core. If the access is an instruction read access, the block transitions to the

[4] Existing code blocks may get written to during JIT compilation, dynamic linking, and self-modification of code. These accesses come to the LLC as *data* writes and are handled as usual like normal data writes.

TABLE II
SIMULATED APPLICATIONS

Suite	Applications	Input/Configuration	Simulation length
PARSEC	bodytrack	sim-medium	Complete ROI
	swaptions	sim-small	
SPLASH-2[3]	barnes	32K particles	Complete ROI
	ocean_cp	514 × 514 grid	
SPEC OMPM2001	314.mgrid	ref input	One charge, one iteration
	316.applu	train input	Six pseudo-time-steps
	324.apsi	train input	One time-step
	330.art	train 2 inputs	Complete parallel section
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions
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 JVM	sunflow, compress	Five operations	Five billion instructions in ROI

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

TABLE III
LLC BLOCK STATES

V	D	State
0	0	Invalid
1	0	Valid, not modified, unowned, not shared
1	1	Valid, modified, unowned, not shared
0	1	Valid, either owned by a core or shared, part of data block used for extended state encoding

TABLE IV
LLC BLOCK EXTENDED STATES

Bit	State
b_0	Dirty
b_1	Pending/Busy
b_2	Exclusively owned ($b_2 = 1$) or shared ($b_2 = 0$)
b_3	Sharer encoding format: If $b_3 = 1$ then bits $b_4, \ldots, b_{3+\lceil \log_2(C) \rceil}$ encode a sharer/owner. If $b_3 = 0$ then bits $b_4, \ldots, b_{3+C}$ encode a C-bit sharer bitvector.

corrupted shared state ($b_2 = 0$); otherwise it transitions to the corrupted exclusive state ($b_2 = 1$). The core id of the requester is recorded using the pointer format ($b_3 = 1$).

A read access to a block in the corrupted exclusive state further changes the state of the block to the corrupted shared state, and the requester obtains the data block from the exclusive owner. A read access to a block in the corrupted shared state leaves the block in the same state, and one of the sharers is elected on-the-fly to supply the data block to the requester. In this case, the critical path of the access increases to three hops (requester to home LLC bank, home LLC bank to the elected sharer, and elected sharer to the requester), instead of two hops in the baseline protocol (LLC would have supplied the data block in the baseline). The sharers are recorded using the bitvector format ($b_3 = 0$).

A read-exclusive access to a block in the corrupted exclusive or corrupted shared state is handled similarly. In the latter case, in addition to electing a sharer to supply the data block, all sharers are invalidated and the LLC block is switched to the corrupted exclusive state. The invalidation acknowledgements are collected at the requester. In this case, the critical path does not increase because even in the baseline, the invalidation acknowledgements from the sharers must be collected at the requester before the request can complete.[5] In the in-LLC protocol, one of these invalidation acknowledgements is of a special type and carries the required data block. An upgrade access to a block in the corrupted shared state invalidates the sharers and the LLC block transitions to the corrupted exclusive state.

An E state eviction (common case for clean private blocks) notification from the private cache hierarchy carries the first $4+\lceil \log_2(C) \rceil$ bits of the evicted data block to the LLC so that the LLC can reconstruct the block. An M state eviction notification from the private cache hierarchy carries the full data block to the LLC, as usual. An S state eviction notification from the private cache hierarchy does not carry any data with it, as in the baseline. In all cases, the evicting core holds the block in a buffer until it receives an acknowledgement from the LLC. This is required to resolve late intervention races. On receiving an eviction notice from the last sharer of a block in S state, the LLC sends a special eviction acknowledgement to the sharer requesting it to send the portion of the block necessary for reconstruction. The sharer supplies the requested portion from the buffer where the block is held.

On eviction of a corrupted dirty block from LLC, the corrupted part of the block is reconstructed by querying either the owner or an elected sharer depending on the extended state of the block. If the block is found dirty in the private cache of the owner, the entire block is sent to the LLC, as usual. All sharers are back-invalidated.

The LLC needs to execute additional writes to the data array for updating the coherence state. These writes are, however, off the critical path and the LLC has ample free write bandwidth to handle these. Also, the coherence action (if any) for a block in the corrupted state (V=0, D=1) can be initiated only after the data block is read out and the first few bits are examined. However, this few cycles of additional delay in initiating the coherence actions for a subset of the shared accesses constitutes a very small percentage of the overall round-trip latency of a private cache miss for the scale of the systems we are dealing with. As a result, this additional delay has negligible impact on the overall performance.

C. Performance Analysis

The in-LLC coherence tracking technique suffers from two shortcomings. First, the read accesses to blocks in corrupted shared state require three transactions in the critical path compared to two transactions in the baseline sparse directory. This

[5] Our simulated system implements sequential consistency and does not support eager-exclusive responses [17], [18].

can be a major performance concern. Second, the reconstruction of the LLC blocks introduces small additional network traffic in the form of the first few bits ($4 + \lceil \log_2(C) \rceil$ or $4 + C$) of a subset of the blocks evicted from the private cache hierarchy.

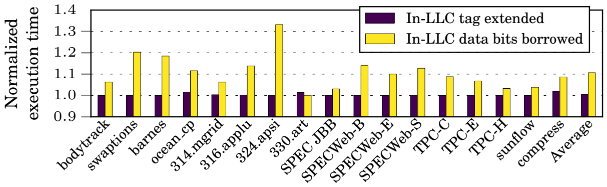

Fig. 4. Performance of in-LLC coherence tracking normalized to a 2× sparse directory.

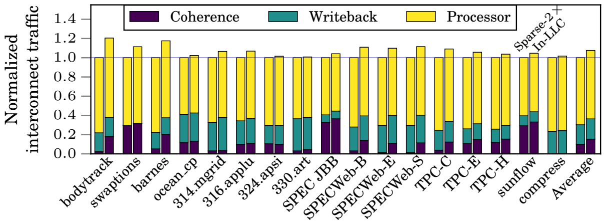

Fig. 5. Interconnect traffic for in-LLC coherence tracking normalized to a 2× sparse directory.

Figure 4 quantifies the execution time of the in-LLC coherence tracking mechanism normalized to a 2× sparse directory. For each application, we evaluate two implementations. The left bar corresponds to a storage-heavy implementation where each LLC block's tag is extended to track coherence. The right bar corresponds to the in-LLC tracking mechanism that we introduced in Sections III-A and III-B. According to Table I, the number of blocks in the LLC is same as the number of entries in a 2× sparse directory. As a result, the storage-heavy implementation delivers similar average performance as the baseline 2× sparse directory. On the other hand, the in-LLC tracking mechanism introduced in this section suffers from an 11% increase in execution cycles, on average. Several applications suffer from more than 10% increase in execution time. For each application, the difference in performance between the two bars arises from the lengthened critical path (three-hop) of the read requests to blocks in the shared corrupted state in the in-LLC tracking mechanism that borrows data bits to maintain coherence information. In the following, we study the performance of this in-LLC coherence tracking mechanism in more detail.

Figure 5 quantifies the interconnect traffic (in bytes) of the in-LLC tracking mechanism normalized to the 2× sparse directory baseline. For each application, the left bar corresponds to the 2× sparse directory baseline and the right bar corresponds to the in-LLC tracking mechanism that borrows data bits to maintain coherence information. Each bar is divided into three segments representing three different types of messages. The private cache misses and their responses constitute the processor messages. The eviction notices from the cores and their acknowledgements constitute the writeback messages. The forwarded requests from the home LLC bank and the corresponding busy-clear messages (if any) coming back to the home LLC bank constitute the coherence messages. On average, the processor and writeback traffic increases by about a percentage each in the in-LLC tracking mechanism. The processor traffic increases due to an increased volume of negative acknowledgements and retries arising from a larger number of LLC blocks being in the busy state waiting to complete forwarded shared read requests. The writeback traffic increases due to inclusion of the first few bits of the evicted block required for LLC block reconstruction in some cases. The coherence traffic increases by more than 5%, on average. This increase is primarily due to the extra forwarded requests arising from the reads to the shared corrupted blocks.

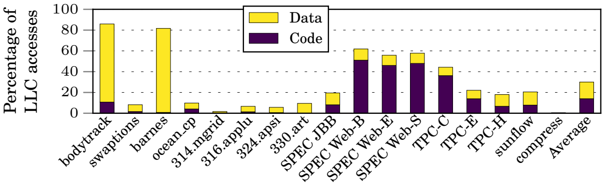

Fig. 6. Percentage of LLC accesses which suffer an increase in critical path.

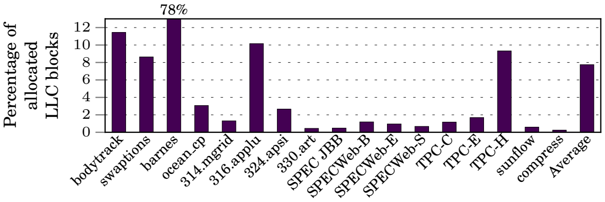

Fig. 7. Percentage of allocated LLC blocks which experience lengthened accesses.

Figure 6 shows, for each application, the percentage of the LLC accesses that require a three-hop transaction in the in-LLC protocol, while the baseline 2× sparse directory could have served these through two-hop transactions. These are essentially read accesses to blocks in the shared corrupted state. On average, 30% of LLC accesses suffer from an increase in the critical path. For some of the commercial applications, among the lengthened accesses, the code accesses are more in population than the data accesses. Figure 7 further shows, for each application, the percentage of the allocated LLC blocks which experience these lengthened accesses. These blocks are a subset of those shown in Figure 2. On average, 8% of the allocated LLC blocks cover all the offending accesses. Barnes is a clear outlier with 78% of the blocks experiencing lengthened accesses. Among the rest, only bodytrack, swaptions, 316.applu, and TPC-H have more than 5% LLC fill population experiencing accesses with lengthened critical path. This result clearly points to a viable sparse directory design that can track this small fraction of LLC blocks and eliminate the performance drawback of the in-LLC protocol. This observation forms the foundation of our tiny directory proposal.

To further understand the extent of sharing experienced by the blocks considered in Figure 7, we introduce the **S**hared **T**hree-hop **R**ead **A**ccess (STRA) ratio. The STRA ratio for an allocated LLC block is the fraction of read accesses to the block which need to be forwarded to a sharer because the state of the block is shared corrupted. All blocks considered in Figure 7 have non-zero STRA ratios and all other blocks have zero STRA ratio. We classify the blocks with non-zero STRA ratios into seven categories $C_1, \ldots, C_7$. The category C_i for $i \in [1, 6]$ includes all LLC blocks with STRA ratio $\in (1 - \frac{1}{2^{i-1}}, 1 - \frac{1}{2^i}]$. The category C_7 includes all the LLC blocks with STRA ratio $\in (1 - \frac{1}{64}, 1]$. Figure 8 shows the distribution of the allocated LLC blocks with non-zero STRA ratios. Figure 9 shows the distribution of the LLC read accesses to shared corrupted blocks based on the category of the involved block. On average, we see that categories C_6 and C_7 account for 54% of these LLC accesses (Average bar in Figure 9), while these two categories cover only 12% of the LLC blocks that source the offending accesses (Average bar in Figure 8). This observation further substantiates the possibility of a tiny directory, which can track the coherence information of this small fraction of blocks.

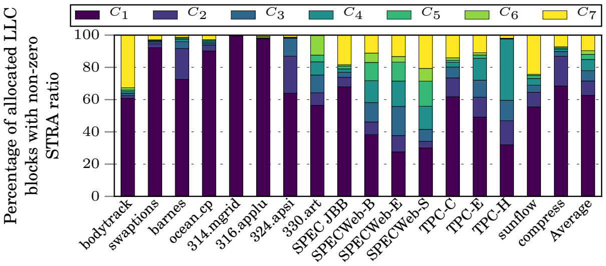

Fig. 8. Distribution of the allocated LLC blocks based on the STRA ratio.

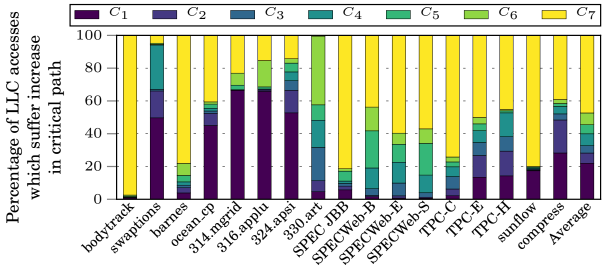

Fig. 9. Distribution of offending LLC accesses based on the accessed block category.

IV. Tiny Directory Proposal

The tiny sparse directory augments the in-LLC coherence tracking mechanism. The goal of the tiny directory design is to track the coherence information pertaining to a subset of blocks with high STRA ratio. However, the small size of the tiny directory makes the selection of the blocks that are tracked in the directory very important. There are two situations in which a block can be considered for being tracked in the tiny directory: (i) when a read request comes for a block which is in the corrupted state, and (ii) when an instruction read request comes for a block in unowned/non-shared/invalid state. As already noted, instruction reads are always responded to in the shared state to accelerate code sharing. In both these situations, if the block is tracked in the tiny directory, the subsequent shared read requests to such a block can be concluded using two-hop transactions.

The tiny directory design consults an allocation policy in these two situations to decide if the requested block's coherence information should be tracked in the tiny directory. If the decision is not to allocate a tiny directory entry for the block, the in-LLC coherence tracking extensions, discussed in Section III-B, are used to track coherence information for the block. On the other hand, if the decision is to allocate a tiny directory entry for the requested block, the LLC block is reconstructed (in case it is in a corrupted state) by forwarding the request to an elected sharer or the owner and asking the elected sharer or the owner to not only forward the block to the requester but also send the corrupted bits of the block to the LLC. The LLC block is switched to a non-corrupted valid state. The coherence state of the block is transferred to the allocated tiny directory entry for further tracking.

On eviction of a tiny directory entry, instead of back-invalidating the sharers, the evicted entry's coherence state is transferred to the corresponding LLC data block and the LLC block transitions to an appropriate corrupted state. If the evicted entry's data block is not present in the LLC (such cases are rare), the sharers are back-invalidated. For the best outcome, it is important to carry out judicious allocations in the tiny directory and minimize the number of pre-mature evictions. We explore two allocation/eviction policies next.

A. Selective Allocation Policies

The selective allocation policies make use of the STRA ratio that the LLC blocks would have experienced in the in-LLC coherence tracking mechanism. In addition to the seven categories ($C_1, \ldots, C_7$) of non-zero STRA ratio, we use C_0 to denote the category of blocks with zero STRA ratio. For estimating the STRA ratio of an LLC block, two six-bit saturating counters, namely STRA Counter (STRAC) and **O**ther **A**ccess **C**ounter (OAC), are maintained for the block. The STRAC is incremented on LLC read accesses which find the block being requested in the shared state (such an access would have resulted in a three-hop critical path in in-LLC coherence tracking). The OAC is incremented on all other LLC accesses (except write-back) to the block. Both the counters of the block are halved when any of the counters has saturated. The STRA ratio estimate for the block is given by the fraction $\frac{STRAC}{STRAC+OAC}$. For the blocks being tracked in the tiny directory, the directory entry is extended by twelve bits to accommodate the two counters. For the LLC blocks in corrupted state, twelve bits are borrowed from the LLC data block to maintain the two counters (this lengthens the corrupted portion by twelve more bits). When the coherence information is transferred between a tiny directory entry and the corresponding LLC data block, both the access counters are also transferred. Once a block returns to the unowned/non-shared state, the counters are reset and the STRA ratio for the block is deemed zero. In the following, we discuss two allocation/eviction policies for the tiny directory.

1) Dynamic STRA Policy: The Dynamic STRA (DSTRA) policy first looks for an invalid way in the tiny directory target set. If there is no such way, it locates the way w with the lowest STRA category (say, C_i) in the target set. If there are multiple ways with the lowest STRA category, the one with the lowest physical way id is selected. Let the STRA category of the block B for which the tiny directory allocation policy is invoked be C_j. The DSTRA policy victimizes the entry w to track block B only if $i < j$. In summary, this policy tries to track a subset of blocks with maximum STRA ratio in the tiny directory. However, one major shortcoming of this policy is that a block belonging to the C_7 STRA category, once tracked in the tiny directory, will occupy a tiny directory entry for a long time until its STRA ratio comes down. This becomes particularly problematic if the block is not accessed for a long time. Our next policy proposal remedies this problem.

2) DSTRA with Generational NRU Policy: The DSTRA with generational not-recently-used policy (DSTRA+gNRU) divides the entire execution into intervals or generations. Each tiny directory entry is extended with two state bits, namely, a reuse (R) bit and an eviction priority (EP) bit. When a tiny directory entry is filled or accessed, the R bit of the entry is set and the EP bit is reset, recording the fact that the entry has been recently accessed and must not be prioritized for eviction in the current interval. At the end of each interval, the tiny directory entries are examined and if an entry's R bit is reset, its EP bit is turned on signifying that the entry can be considered for eviction in the next interval. The R bits of all entries are gang-cleared at the beginning of each interval signifying the start of a new generation of reuses.

The DSTRA+gNRU policy proceeds similarly to the DSTRA policy and selects a way w with the lowest STRA category (C_i) in the target set. If there are multiple ways with the lowest STRA 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 STRA category of the block B for which the tiny directory allocation policy is invoked be C_j. The DSTRA+gNRU policy victimizes the entry w to track block B only if one of the two following conditions is met: (i) $i < j$, (ii) $i == j$ and the EP bit of w is set. The second condition effectively creates an avenue for replacing useless entries of a certain STRA category.

The length of a generation needs to be chosen carefully. We set the generation length to the interval between two consecutive reuses to a tiny directory entry, averaged across all entries. 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. Each tiny directory slice attached to an LLC bank maintains a ten-bit counter T which is incremented by one every 4K cycles (measured using a twelve-bit counter). Each tiny directory entry is extended by ten bits to record the value of counter T whenever the entry is accessed. On an access to a tiny directory entry, the last recorded value of counter T in the tiny directory entry (T_{last}) is compared with the current value of counter T in the slice ($T_{current}$). If $T_{last} < T_{current}$, the difference between $T_{current}$ and T_{last} is added to a counter A. The counter A is maintained per tiny directory slice and records the accumulated time between two consecutive accesses to a tiny directory entry. Another counter B maintained per tiny directory slice records the number of values added to counter A. At any point in time, the generation length used by a tiny directory slice is estimated as $\frac{A}{B}$. At the beginning of an interval, this value is copied to a generation length counter, 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.

B. Introducing Robustness: Spilling into LLC

The tiny directory is only capable of identifying and tracking the coherence state of a subset of blocks that are most likely to suffer in terms of lengthened critical path of shared read accesses in the in-LLC coherence tracking mechanism. Since the size of this performance-critical shared working set of an application is not known beforehand and may even vary during execution, it is impossible to design an optimally-sized tiny directory that can offer robust and reliable performance for a wide range of applications. To address this problem, we augment the tiny directory design with the option of selectively spilling a subset of coherence tracking entries into the LLC. A spilled coherence tracking entry occupies a tag and the corresponding data block in the LLC. It is different from the data block for which coherence is being tracked. As a result, a fundamental challenge in enabling a coherence tracking entry spill policy is to ensure that the volume of LLC misses does not increase due to the pressure of the spilled tracking entries. Section IV-B1 discusses the organization and maintenance of a spilled coherence tracking entry. Section IV-B2 describes the selective spill policy, which identifies the coherence tracking entries eligible for spilling.

1) Organization of Spilled Entries: The reason for enabling spilling of coherence tracking entries into the LLC is to avoid lengthening the critical path of shared read accesses when the tiny directory is unable to track all such shared blocks. As a result, a coherence tracking entry E_B of a block B can be spilled into the LLC only if B is currently in the shared state. A spilled coherence tracking entry E_B is allocated in a way in the same set as block B. Since B and E_B have the same tag, this allocation decision guarantees that in an LLC set, there can be at most two tag matches on a lookup. To distinguish between the block B and the block holding E_B, we use the state (V=0, D=1) for the spilled tracking entries. B cannot be in a corrupted state and hence, it will have V=1. The LLC replacement policy always victimizes a spilled coherence tracking entry E_B before the corresponding block B. This is ensured by the LRU position update policy of the LLC: first, we move E_B to the MRU position and then B to the MRU position whenever B and E_B are accessed. When E_B is chosen as a victim, the coherence information is transferred to B and B switches to the corrupted shared state.

If an LLC lookup indicates two tag matches, we know that the one with state V=1 corresponds to the data block and the other one is the spilled coherence tracking entry for the block. On the other hand, if the lookup returns a single tag match, the state of the matched tag decides if the block is in a corrupted state (V=0, D=1) or not (V=1). As usual, the tiny directory is always looked up in parallel with the LLC and a tiny directory hit indicates that the coherence of the block is being tracked in the tiny directory.

On an access to a data block B, if the coherence tracking entry E_B is also in the same set, the two blocks have to be read out sequentially. To avoid lengthening the critical path, on a read request, we read out the data block B first and respond to the requester (recall that spilling is allowed only for blocks in the shared state). Next, we read out E_B and update the coherence tracking information. On a read-exclusive request, we read out E_B first and send out the invalidations and also ask an elected sharer to forward the data block to the requester. On an upgrade request, we read out E_B first and send out the invalidations. For both read-exclusive and upgrade requests, the block E_B is invalidated and the coherence information is transferred to B, which now switches to the corrupted exclusive state.

2) Selective Spill Policy: The selective spill policy for coherence tracking entries determines if the coherence information of a block can be tracked by spilling it in the LLC. This policy is invoked in two situations: (i) when the tiny directory's allocation policy declines to track the coherence information of a requested block in the tiny directory, and (ii) eviction of a tiny directory entry corresponding to a block in the shared state. If the policy decision is not to spill in the LLC, the in-LLC coherence tracking extensions are used to track the coherence information of the involved block. If the policy decision is to spill in the LLC, a way is allocated in the same LLC set as the involved block to track the coherence information of the block. In this case, if the involved block is found in a corrupted state in the LLC, it is reconstructed following the reconstruction procedure discussed already and the block transitions to a non-corrupted valid state (V=1).

Whenever the spill policy is invoked, its goal is to allow spilling of coherence tracking entries for blocks with high STRA ratio. At the same time, the spill policy must keep a check on the LLC miss rate for data blocks. Let C_j be the STRA category of the block which is trying to spill its coherence tracking entry in the LLC under one of the two aforementioned situations when the spill policy is invoked. We formulate the selective spill policy design problem as follows. The selective spill policy should dynamically determine the highest STRA category C_i such that the coherence tracking entries for the blocks with STRA category C_j with $j \geq i$ can be spilled in the LLC whenever needed while guaranteeing that the LLC miss rate for data blocks increases by no more than a pre-defined value of δ. The value δ represents the tolerance limit for LLC miss rate. Each LLC bank independently implements this policy and determines a suitable C_i for the bank. The index i of this computed C_i for an LLC bank will be referred to as the STRA spill threshold category index of the bank and this selective spill policy will be referred to as the Dynamic Spill policy. We discuss its implementation in the following.

In each LLC bank, sixteen sets are kept aside that do not admit any spilled coherence tracking entries. These sets are used to estimate the LLC bank's miss rate without spilling ($MR_{no-spill}$). The remaining sets exercise spilling for STRA categories C_j such that $j \geq i$, given a dynamically computed STRA spill threshold category index i for the LLC bank. From these sets, the LLC bank's miss rate with spilling (MR_{spill}) can be determined. We define a window of observation for an LLC bank as 8K accesses (except writebacks) to the bank. At the end of each observation window, if $MR_{spill} \leq MR_{no-spill} + \delta$ is satisfied (meaning that due to spilling, the LLC bank's miss rate increases by no more than δ), the STRA spill threshold category index i is decreased by one in that bank so that a bigger volume of spills can be admitted in the next observation window. On the other hand, if $MR_{spill} \leq MR_{no-spill} + \delta$ is not satisfied, i for the bank is increased by one so that the spill volume can be reduced. We note that the value of i saturates at zero and seven on the two sides of the admissible range.

The aforementioned policy for dynamically determining the STRA spill threshold category index may lead to oscillations in the index value around the convergence point unless the index i saturates to zero or seven. Such oscillations are easy to detect and the STRA spill threshold category index can be fixed to one of the two oscillation values such that $MR_{spill} \leq MR_{no-spill} + \delta$ is satisfied. However, fixing the index value to avoid oscillation may cause the state of the algorithm to get stuck at that index value leading to lost opportunity of spilling more in certain phases of execution. Coming out of such a state will require complex mechanisms to detect phase changes when a new lower index value can be tried. This is complicated by the fact that MR_{spill} for a certain STRA spill threshold category index cannot be determined by sampling a few LLC sets (like the way we determine $MR_{no-spill}$) because the spill volume distribution is non-uniform and highly skewed toward the LLC sets that accommodate shared blocks. We, however, note two important aspects about this oscillation. First, if an oscillation at all happens, it is restricted to the few phases of execution that experience high to moderate volumes of spilling because small amount of spilling cannot change the LLC miss rate much. Second, since the length of the observation window is quite large (8K accesses per bank × 128 banks or 1M LLC accesses on average), the oscillation in the index value happens at a reasonably slow rate. At the end, to keep the design simple, we decide to use our Dynamic Spill policy without any change to arrest oscillation. Our evaluation of this policy shows that even with the possibility of such oscillations in certain phases, the increase in the LLC miss rate due to spilling never exceeds the guarantee offered by the value of δ.

Selection of an appropriate δ is important for the success of the proposed spill policy. We define the overall STRA ratio of an application as the number of LLC reads to blocks in the shared state over the total number of LLC accesses (except writebacks). In general, if an application has a very low LLC miss rate, it may not be able to tolerate a large increase in LLC miss rate because such applications are typically very latency-sensitive. On the other hand, if an application is undergoing a phase of overall high STRA ratio, it may be possible to convert a larger proportion of LLC hits to misses and gain in terms of shared read hit latency by spilling more. Within each LLC bank, we measure the miss rate and the overall STRA ratio. At the end of each window of observation, each LLC bank independently classifies the running application into four possible categories: (A) LLC bank's miss rate is at least 10% and STRA ratio is at least 0.4, (B) LLC bank's miss rate is at least 10% and STRA ratio is below 0.4, (C) LLC bank's miss rate is below 10% and STRA ratio is at least 0.4, and (D) LLC bank's miss rate is below 10% and STRA ratio is below 0.4. At the beginning of each observation window, each LLC bank independently decides the value of δ to be used in that bank depending on the category of the application observed during the last window: $\delta_A = \frac{1}{4}$, $\delta_B = \frac{1}{32}$, $\delta_C = \frac{1}{16}$, $\delta_D = \frac{1}{32}$. The categories with higher STRA ratio are assigned higher δ values while keeping the miss rate profile in mind. These values may require tuning depending on the system configuration.

C. Coherence Processing Latency at LLC

Among the coherence processing paths traversed by the tiny directory proposal at the LLC bank controller, there are two situations where the critical path gets slightly lengthened compared to the baseline. Both the cases arise from accessing a block in the corrupted state. If the accessed block is in the corrupted shared state, the LLC tag and data must be accessed serially followed by decoding of the coherence state from the data block before responding to the requester. In the baseline, the critical path through the LLC bank controller for accessing such a block would involve only the serial access of the LLC tag and data (overlapped with sparse directory access). In this case, we charge one extra cycle of LLC latency for the tiny directory implementation accounting for the coherence state decoding overhead. If the accessed block is in the corrupted exclusive state, the tiny directory proposal must access the LLC tag and data serially and then decode the coherence state before forwarding the request to the owner. In the baseline, the critical path through the LLC bank controller for accessing such a block would involve only the LLC tag access latency overlapped with the sparse directory lookup latency. In this case, the tiny directory proposal suffers from two additional cycles of LLC data access latency (see Table I) followed by one cycle of coherence state decoder latency. We model all these additional latencies in our evaluation.

V. Simulation Results

We evaluate our proposal in this section for four different tiny directory sizes: $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$. The $\frac{1}{32}\times$ and $\frac{1}{64}\times$ sizes have respectively 64 and 32 entries per tiny directory slice attached to an LLC bank. Both these sizes exercise eight-way set-associative directory slices. The $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes have respectively 16 and 8 entries per tiny directory slice and exercise fully-associative configurations. Each directory entry has a size of 155 bits (128-bit sharer vector, 12 bits for STRAC and OAC, 10 bits for the timestamp counter used to estimate the generation length in the gNRU policy, two bits for R and EP states used by the gNRU policy, one bit for pending/busy transient state, and two coherence state bits for tracking invalid, exclusively owned and shared states). Additionally, each directory entry has a tag of the following lengths (we assume 48-bit physical address): 32 bits for $\frac{1}{32}\times$, 33 bits for $\frac{1}{64}\times$, 35 bits for $\frac{1}{128}\times$ and $\frac{1}{256}\times$. As a result, the total storage investment for coherence tracking across all 128 slices is as follows: 187 KB for $\frac{1}{32}\times$, 94 KB for $\frac{1}{64}\times$, 47.5 KB for $\frac{1}{128}\times$, and 23.75 KB for $\frac{1}{256}\times$.

Figures 10 and 11 evaluate our proposal for $\frac{1}{32}\times$ and $\frac{1}{64}\times$ sizes, respectively. These figures quantify the percentage increase in execution cycles compared to a 2× directory. For each tiny directory size, we show the results with the DSTRA allocation policy, DSTRA+gNRU allocation policy, and DSTRA+gNRU augmented with dynamic spilling (DynSpill) of coherence tracking entries. For the $\frac{1}{32}\times$ size (Figure 10), both DSTRA and DSTRA+gNRU policies are, on average, within 1% of the performance of 2× directory; when dynamic spilling is enabled, the gap reduces to 0.5%. Referring back to Figure 4, we note that the in-LLC coherence tracking mechanism is 11% worse than the 2× directory. Introduction of a tiny directory bridges this gap.

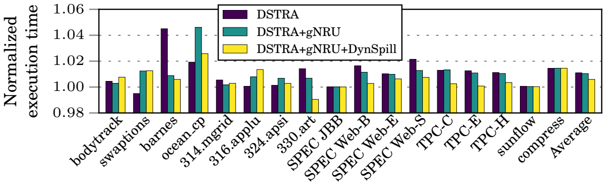

Fig. 10. Performance of $\frac{1}{32}\times$ tiny directory normalized to a sparse 2× directory.

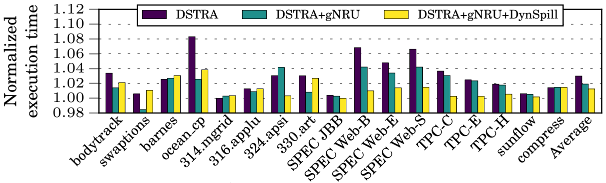

Fig. 11. Performance of $\frac{1}{64}\times$ tiny directory normalized to a sparse 2× directory.

For the $\frac{1}{64}\times$ size (Figure 11), the gNRU-assisted allocation policy begins to gain in importance in some of the applications (ocean_cp and SPECWeb). On average, the DSTRA policy has 3% higher execution cycles compared to the 2× directory, while the DSTRA+gNRU policy is only 2% away from the 2× directory. Dynamic spilling further brings this gap down to 1%.

Referring back to the discussion related to Figure 3, we note that a skew-associative directory that tracks only shared blocks suffers from a 12% slowdown for the $\frac{1}{64}\times$ size compared to a 2× directory, on average. Our set-associative tiny directory without dynamic spilling at this size performs far better underscoring the success of the DSTRA and the DSTRA+gNRU policies which capture a critical subset of the shared blocks. Referring back to Figure 7, we note that this critical subset accounts for 78% of all allocated LLC blocks for barnes. Even for this application, our tiny directory proposal is able to capture the instantaneous working set of these critical blocks and deliver performance close to a 2× directory.

Figures 12 and 13 evaluate our proposal for $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes, respectively. At these two sizes, the gNRU policy gains further in importance in several applications. On average, for the $\frac{1}{128}\times$ size, the DSTRA and the DSTRA+gNRU policies have 6% and 5% higher execution cycles compared to the 2× directory. The dynamic spill policy assumes significant importance at these small directory sizes and brings down the gap between our proposal and the 2× directory to 1%. Referring back to Figure 3, we note that a sparse directory that tracks only shared blocks suffers from a 28% slowdown for the $\frac{1}{128}\times$ size compared to a 2× directory, on average. Our tiny directory proposal successfully wipes away this performance loss.

For the $\frac{1}{256}\times$ size (Figure 13), the DSTRA and the DSTRA+gNRU policies have 8% and 6% higher execution cycles compared to the 2× directory, on average. Dynamic spilling reduces this gap to 1%. In summary, our tiny directory proposal offers robust performance staying within a percentage of a sparse 2× directory as the tiny directory size is varied between $\frac{1}{32}\times$ and $\frac{1}{256}\times$ (187 KB to 23.75 KB).

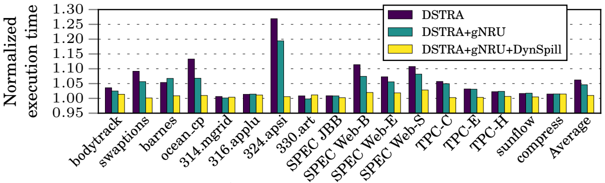

Fig. 12. Performance of $\frac{1}{128}\times$ tiny directory normalized to a sparse 2× directory.

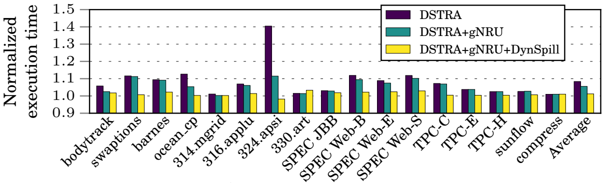

Fig. 13. Performance of $\frac{1}{256}\times$ tiny directory normalized to a sparse 2× directory.

A. Analysis of Performance

The main purpose of the tiny directory proposal is to eliminate most of the additional three-hop transactions that the in-LLC coherence mechanism introduced. When these three-hop transactions get replaced by the two-hop transactions as in the sparse 2× directory, the performance is expected to be similar to the 2× directory. Referring back to Figure 6, we note that the percentage of LLC accesses that suffer from an increased critical path because they get extended to three-hop transactions in the in-LLC coherence tracking mechanism is 30% on average. To confirm that our proposal is able to address this problem successfully, Figures 14 and 15 show the percentage of the LLC accesses that suffer from an increase in the critical path for a $\frac{1}{32}\times$ tiny directory system and a $\frac{1}{256}\times$ tiny directory system, the two extreme points of our size spectrum. For a $\frac{1}{32}\times$ tiny directory, the DSTRA and the DSTRA+gNRU policies have only 3% and 2% such LLC accesses on average. The dynamic spill policy brings this average to under 1%. For a $\frac{1}{256}\times$ tiny directory, this percentage increases significantly for the DSTRA and the DSTRA+gNRU policies. These policies experience 23% and 20% such LLC accesses respectively (still lower than in-LLC mechanism), while the dynamic spill policy successfully brings this average down to only 4%. These small residual extra three-hop transactions cause a percent loss in performance.

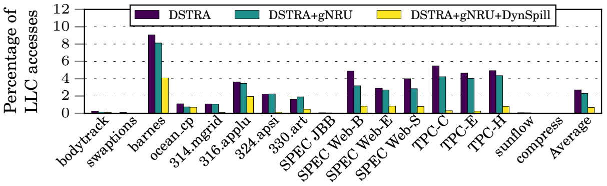

Fig. 14. Percentage of LLC accesses which suffer from an increase in critical path in a $\frac{1}{32}\times$ tiny directory.

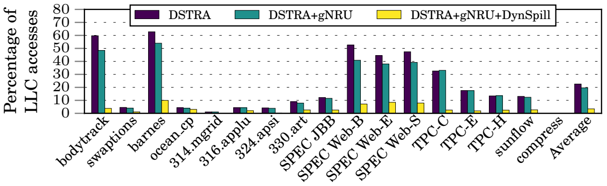

Fig. 15. Percentage of LLC accesses which suffer from an increase in critical path in a $\frac{1}{256}\times$ tiny directory.

The success of the tiny directory in reducing the number of extra three-hop transactions depends on the number of hits that a tiny directory entry enjoys. Figure 16 shows the number of tiny directory hits for the DSTRA+gNRU policy normalized to the DSTRA policy for all the four directory sizes. As the directory size decreases from $\frac{1}{32}\times$ to $\frac{1}{256}\times$, the gNRU policy gains in importance. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, the DSTRA+gNRU policy offers, respectively, 3%, 12%, 23%, and 39% more directory entry hits compared to the DSTRA policy. The biggest beneficiaries of the gNRU policy are bodytrack, swaptions, barnes, ocean_cp, 330.art, and SPECWeb. The primary advantage of the gNRU policy is that it quickly removes the useless directory entries, which the DSTRA policy would have retained for a long time. This creates room for more useful directory entries to be tracked. Figure 17 validates this behavior by quantifying the number of allocations in the tiny directory experienced by the DSTRA+gNRU policy normalized to the DSTRA policy for all the four directory sizes. As the directory size decreases from $\frac{1}{32}\times$ to $\frac{1}{256}\times$, the gNRU policy allows a much larger number of directory fills to take place, thereby significantly increasing the effective coverage of the tiny directory. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, the DSTRA+gNRU policy observes, respectively, 2×, 7×, 50×, and 74× more directory fills compared to the DSTRA policy. Figure 18 quantifies the average number of hits enjoyed by a directory entry before getting replaced for the DSTRA+gNRU policy. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, this number is 59.5, 46.1, 16.6, and 17.5, respectively. This result confirms that the directory entries tracked by the DSTRA+gNRU policy are indeed important. They enjoy a significant number of hits before getting replaced even for the smallest size.

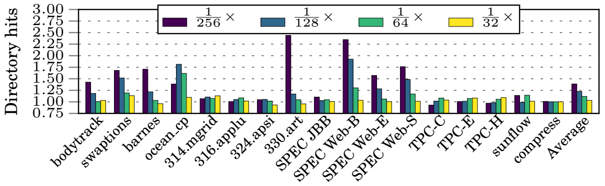

Fig. 16. Hits in tiny directory with the DSTRA+gNRU policy normalized to the DSTRA policy.

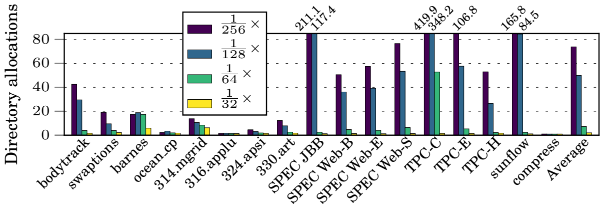

Fig. 17. Allocations in tiny directory with the DSTRA+gNRU policy normalized to the DSTRA policy.

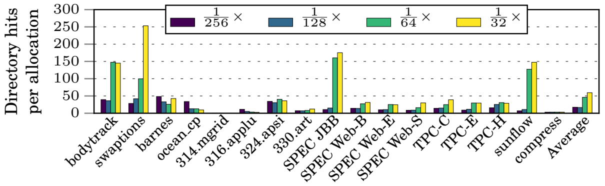

Fig. 18. Hits per allocation in tiny directory with the DSTRA+gNRU policy.

Next, we analyze our dynamic spill policy which we have shown to be highly robust across the board. There are two aspects of the dynamic spill policy that we analyze. Figure 19 shows the percentage of the LLC accesses which are able to avoid increase in critical path because of spilled directory entries when using the DSTRA+gNRU+DynSpill policy. These are essentially read accesses to the blocks, the coherence tracking entries of which are spilled in the LLC. Without these spilled entries, these accesses would get extended to three-hop transactions because the data block would have been in the corrupted shared state. The percentage of such LLC accesses increases significantly as the tiny directory size drops. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, 2%, 5%, 11%, and 16% LLC accesses benefit from spilling. The biggest beneficiaries are bodytrack, barnes, SPECWeb, and TPC.

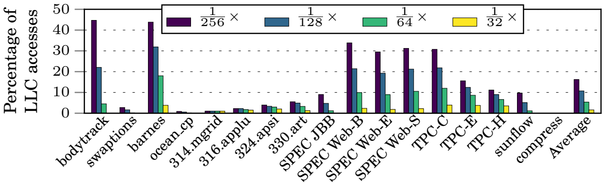

Fig. 19. Percentage of LLC accesses which are able to avoid increase in critical path because of spilled directory entries in the LLC when using the DSTRA+gNRU+DynSpill policy.

The second aspect of the spill policy is its influence on the LLC miss rate. We are particularly interested in the behavior of the applications that already have high LLC miss rates in the baseline. For example, the applications with more than 10% LLC miss rate include ocean_cp (35% LLC miss rate), 314.mgrid (78%), 324.apsi (12%), 330.art (63%), SPECWeb-B (14%), SPECWeb-E (19%), and SPECWeb-S (18%). Our Dynamic Spill policy guarantees an upper bound on the LLC miss rate increase through the δ values. Figure 20 shows the increase in LLC miss rate when using the DSTRA+gNRU+DynSpill policy relative to the sparse 2× directory. As the tiny directory size decreases, the LLC miss rate increases very slowly. Only 316.applu and 330.art show more than 1% increase in the LLC miss rate compared to the 2× directory. Across the board, the maximum increase in the LLC miss rate due to spilling is 2.1% experienced by 316.applu when operating with a $\frac{1}{256}\times$ tiny directory. We note that this is within the smallest δ (the guaranteed upper bound on LLC miss rate increase) that we use (Section IV-B2). The average increase in the LLC miss rate is under 0.5% for all directory sizes.

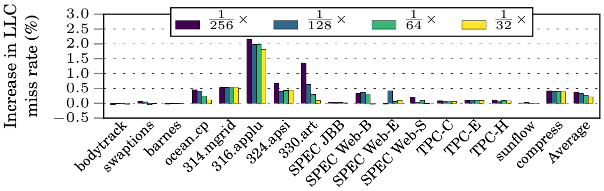

Fig. 20. Increase in LLC miss rate due to spilling when using DSTRA+gNRU+DynSpill policy compared to a 2× sparse directory.

To further confirm that our proposal continues to offer robust performance for smaller LLC capacities, we evaluate our proposal in a configuration where the entire cache hierarchy is halved in terms of the number of sets (the capacity ratio between different levels is maintained) i.e., the shared LLC capacity is 16 MB in both the baseline and our proposal. In this configuration, compared to a sparse 2× directory, the DSTRA+gNRU and DSTRA+gNRU+DynSpill policies experience an average increase of 7% and 1% execution cycles for a $\frac{1}{128}\times$ tiny directory (eight entries fully-associative per slice) where spilling is quite prevalent.

B. Energy Comparison

We use CACTI [21] (distributed with McPAT [22]) to compute the dynamic and leakage energy consumed by the LLC and the sparse directory for 22 nm nodes. Figure 21 shows the dynamic, leakage, and total energy of the LLC and the sparse directory for the baseline configurations (from 2× to $\frac{1}{16}\times$) normalized to the $\frac{1}{256}\times$ tiny directory exercising the DSTRA+gNRU+DynSpill policy. We have also shown the $\frac{1}{128}\times$ tiny directory in the figure. Additionally, the figure includes the trends in the execution cycles as well. As the baseline sparse directory size decreases, the execution cycles monotonically increase, as expected. The dynamic, leakage, and total energy expense first decreases as the baseline directory size shrinks. However, beyond $\frac{1}{4}\times$ size of the baseline directory, the energy expense increases quickly due to increasing execution cycles. Compared to the $\frac{1}{256}\times$ tiny directory, the baseline dynamic energy is much lower. The extra dynamic energy consumption in the tiny directory arises primarily from the additional LLC writes that need to be done to update the coherence information in the corrupted and the spilled entries. On the other hand, the leakage and total energy expense is much lower in the tiny directory due to the drastically reduced size of the directory. The data array of a 2× directory is 8 MB in capacity compared to 47.5 KB and 23.75 KB total sizes of the $\frac{1}{128}\times$ and $\frac{1}{256}\times$ tiny directories. Overall, compared to the baseline sparse 2× directory, our proposal saves 17% and 16% of total LLC and directory energy for the $\frac{1}{128}\times$ and $\frac{1}{256}\times$ tiny directory sizes, respectively. The baseline $\frac{1}{4}\times$ sparse directory configuration comes closest to the $\frac{1}{256}\times$ tiny directory in terms of total energy expense (4% more than the $\frac{1}{256}\times$ tiny directory), but requires 1 MB space for its directory data array and performs 2.5% worse than the $\frac{1}{256}\times$ tiny directory.

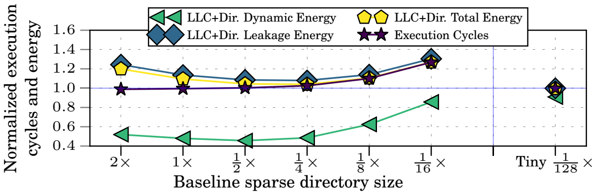

Fig. 21. Execution cycles and energy normalized to the $\frac{1}{256}\times$ tiny directory exercising the DSTRA+gNRU+DynSpill policy.

C. Comparison to Related Proposals

Recent proposals have tried to reduce the number of sparse directory entries by addressing the overhead of tracking the private blocks. These contributions were reviewed in Section I-A. By comparing Figure 3 with Figures 10, 11, and 12 we have already shown that our proposal performs much better than a sparse directory that tracks only shared blocks. None of the recent proposals that try to reduce the overhead of tracking the private blocks can perform better than the ideal sparse directory that tracks only shared blocks. Nonetheless, for completeness, we evaluate the state-of-the-art multi-grain directory (MgD) [47] and the Stash directory [13]. The MgD invests just one directory entry for a private region of size 1 KB, thereby saving significantly on the overhead of tracking the private blocks. The Stash directory does not track private blocks after the corresponding directory entries are evicted and later if such a block gets shared, it resorts to broadcast to reconstruct the directory entry. Figure 22 evaluates a skew-associative MgD for four sizes ($\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$) and an eight-way set-associative Stash directory for $\frac{1}{32}\times$ size. Compared to a 2× sparse directory, the MgD proposal suffers from a 0.1%, 8%, 29%, and 63% increase in average execution cycles for $\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes, respectively. The Stash directory at $\frac{1}{32}\times$ size performs 41% worse than the 2× directory on average. We find that the Stash directory is able to save a significant volume of the private cache misses because it does not back-invalidate the private blocks on sparse directory eviction. However, the broadcast traffic becomes a major bottleneck in this proposal, particularly for the scale of the systems we are considering. In comparison, our proposal exercising directory sizes between $\frac{1}{256}\times$ and $\frac{1}{32}\times$ performs within 1% of a 2× sparse directory.

VI. Summary and Future Directions

We have presented a novel design to track coherence information within a chip-multiprocessor. The design allows us to significantly scale down the traditional sparse directory size. Our proposal has three major components. First, it tracks the private block owner by borrowing a few bits of the last-level cache data block. Second, it employs a tiny directory that tracks the coherence information of a critical subset of the blocks belonging to the shared working set. Third, if the tiny directory is too small to track all the critical shared blocks, the proposal employs dynamic selective spilling of the coherence tracking entries into the LLC. Any remaining block is tracked by borrowing a few

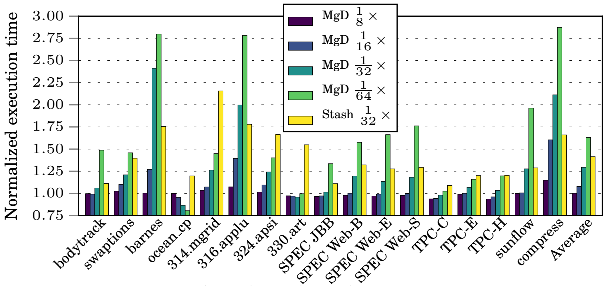

Fig. 22. Performance of $\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$ multi-grain directory (MgD) and $\frac{1}{32}\times$ Stash directory normalized to a sparse 2× directory.

bits of the LLC data block. The simulation results on a 128-core system for a wide range applications show that our proposal operating with tiny directories of size ranging between $\frac{1}{32}\times$ and $\frac{1}{256}\times$ performs within 1% of a system with a 2× sparse directory.

The application of the tiny directory proposal to inter-socket coherence tracking in a multi-socket environment is the next natural step to explore. We believe that this could be a promising future direction for significantly reducing the space invested to maintain the inter-socket coherence directory.

References

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A New Scalable Directory Architecture for Large-scale Multiprocessors. In *HPCA* 2001.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. In *IEEE TPDS*, January 2005.

[3] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *ISCA* 1988.

[4] M. Alisafaee. Spatiotemporal Coherence Tracking. In *MICRO* 2012.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT* 2008.

[6] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. In *IEEE TC*, December 1978.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *ASPLOS* 1991.

[8] Y. Chang and L. Bhuyan. An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors. In *ICPP* 1996.

[9] G. Chen. SLiD – A Cost-effective and Scalable Limited-directory Scheme for Cache Coherence. In *PARLE* 1993.

[10] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT* 2011.

[11] J. H. Choi and K. H. Park. Segment Directory Enhancing the Limited Directory Cache Coherence Schemes. In *IPDPS* 1999.

[12] B. A. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *ISCA* 2011.

[13] S. Demetriades and S. Cho. Stash Directory: A Scalable Directory for Many-core Coherence. In *HPCA* 2014.

[14] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang. Building Expressive, Area-efficient Coherence Directories. In *PACT* 2013.

[15] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-core Systems. In *HPCA* 2011.

[16] A. Garcia-Guirado, R. Fernandez-Pascual, and J. M. Garcia. ICCI: In-cache Coherence Information. In *IEEE TC*, April 2015.

[17] K. Gharachorloo. Memory Consistency Models for Shared Memory Multiprocessors. *WRL Research Report 95/9*, December 1995.

[18] K. Gharachorloo, M. Sharma, S. Steely, and S. vanDoren. Architecture and Design of AlphaServer GS320. In *ASPLOS* 2000.

[19] S. Guo, H. Wang, Y. Xue, D. Li, and D. Wang. Hierarchical Cache Directory for CMP. In *Journal of Computer Science and Technology*, March 2010.

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

[21] 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/.

[22] 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/.

[23] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Distributed Directory Scheme: Scalable Coherent Interface. In *IEEE Computer*, June 1990.

[24] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel. WAYPOINT: Scaling Coherence to Thousand-core Architectures. In *PACT* 2010.

[25] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-core Cache-coherent Processor with On-chip Optical Network. In *PACT* 2010.

[26] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA* 1997.

[27] Y. Li, R. G. Melhem, and A. K. Jones. Practically Private: Enabling High Performance CMPs through Compiler-assisted Data Classification. In *PACT* 2012.

[28] Y. Maa, D. Pradhan, and D. Thiebaut. Two Economical Directory Schemes for Large-scale Cache Coherent Multiprocessors. In *ACM SIGARCH Computer Architecture News*, September 1991.

[29] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. In *CACM*, July 2012.

[30] S. S. Mukherjee and M. D. Hill. An Evaluation of Directory Protocols for Medium-scale Shared Memory Multiprocessors. In *ICS* 1994.

[31] H. Nilsson and P. Stenstrom. The Scalable Tree Protocol – A Cache Coherence Approach for Large-scale Multiprocessors. In *IPDPS* 1992.

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

[33] A. Ros and S. Kaxiras. Complexity-effective Multicore Coherence. In *PACT* 2012.

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

[35] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *HPCA* 2012.

[36] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO* 2010.

[37] S. Shukla and M. Chaudhuri. Pool Directory: Efficient Coherence Tracking with Dynamic Directory Allocation in Many-core Systems. In *ICCD* 2015.

[38] R. Simoni and M. Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Shared Memory Multiprocessing* 1991.

[39] R. T. Simoni, Jr. Cache Coherence Directories for Scalable Multiprocessors. *PhD dissertation*, Stanford University, 1992.

[40] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *ASPLOS* 2013.

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

[42] D. Wallach. PHD: A Hierarchical Cache Coherent Protocol. *Ph.D. dissertation*, MIT, 1992.

[43] W.-D. Weber. Scalable Directories for Cache-coherent Shared-memory Multiprocessors. *Ph.D. dissertations*, Stanford University, 1993.

[44] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *ASPLOS* 1989.

[45] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA* 1995.

[46] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Select-Directory: A Selective Directory for Cache Coherence in Many-core Architectures. In *DATE* 2015.

[47] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain Coherence Directories. In *MICRO* 2013.

[48] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *MICRO* 2009.

[49] L. Zhang, D. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin. SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors. In *PACT* 2014.

[50] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *PACT* 2011.

[51] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *PACT* 2010.