Zero Directory Eviction Victim: Unbounded Coherence Directory and Core Cache Isolation

Mainak Chaudhuri
Department of CSE, Indian Institute of Technology Kanpur
mainakc@cse.iitk.ac.in

***Abstract*—A directory structure is traditionally employed for tracking coherence information of the privately cached blocks in a cache-coherent chip-multiprocessor (CMP). The eviction of a directory entry necessarily invalidates the privately cached copies of the block that the evicted entry was tracking. These forced *directory eviction victims* pose two major challenges. First, with decreasing directory size, the volume of these victim blocks increases significantly causing performance degradation. As a result, sizing the directory remains an important challenge. Second, the tight coupling between the directory evictions and the private cache contents can be exploited to launch timing-based side-channel attacks, as has been demonstrated recently. The existing solutions to the first problem allow reducing the directory capacity only up to a certain extent before the performance starts degrading. The existing mitigation technique for the security vulnerability avoids generation of only a certain specific subset of directory victims.**

In this paper, we present the *Zero Directory Eviction Victim* (ZeroDEV) coherence protocol and accompanying novel mechanisms that *guarantee* freedom from invalidations arising from directory victims, thereby completely isolating the private core caches from the coherence directory evictions. This is the first fully hardwired design proposal that enables a practically unbounded coherence directory which, to the core caches in a CMP, appears to never evict a live entry. Unlike the prior proposals that have completely eliminated the directory and the coherence information eviction victims in a multi-/many-core CMP, our proposal does not require any operating system or application software changes. Our proposal, instead, repurposes the on-die last-level cache (LLC) space for holding the evicted directory entries and engineers a novel mechanism to handle directory entry eviction from the LLC without generating any invalidation to the private core caches. The ZeroDEV protocol evaluated on multi-threaded and multi-programmed workloads for inclusive and two popular non-inclusive CMP cache hierarchy designs performs within 1-2% of a well-provisioned traditional baseline. Importantly, as an additional benefit of eliminating directory eviction victims and utilizing the large on-die LLC for caching directory entries, we show that our proposal does not need any dedicated directory structure at all for certain classes of CMP cache hierarchy designs while maintaining the performance level and continuing to guarantee complete isolation of the core caches from directory entry eviction.

***Index Terms*—chip-multiprocessor; cache coherence; sparse directory; directory eviction victim.**

I. Introduction

The coherence directory structure is an integral part of the scalable cache coherence protocols. Within a chip-multiprocessor (CMP), this structure typically takes the form of a sparse directory [16], [27] organized as a tagged set-associative cache. A sparse directory entry keeps track of the coherence state and the location(s) of a block that is cached privately by at least one of the processor cores. The sparse directory entry corresponding to a block is freed when all private copies of the block are evicted from the processor cores. The eviction of a live sparse directory entry must invalidate all privately cached copies of the block the directory entry was tracking. We refer to these private cache blocks invalidated due to directory entry eviction as the directory eviction victims (DEVs). Figure 1 illustrates an example where two blocks B1 and B2 are tracked by sparse directory entries E1 and E2. B1 is cached in cores C0 and C1, while B2 is cached only in core C1. When new directory entries E3 and E4 are allocated in the sparse directory, they evict E1 and E2 respectively. The eviction of E1 generates invalidations to the copies of B1 cached in C0 and C1, while the eviction of E2 generates an invalidation to the copy of B2 cached in C1. These invalidations, in turn, generate the DEVs which are the invalidated blocks i.e., two copies of B1 and one copy of B2.

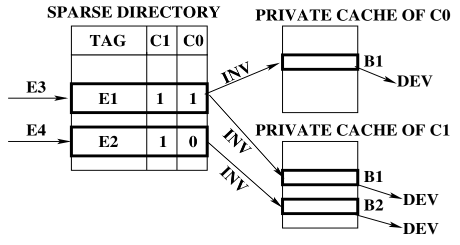


Fig. 1. An example of directory eviction victims (DEVs).

Appropriate sizing of the sparse directory plays a very important role in determining the CMP performance as the volume of DEVs has an inverse relationship with the number of sparse directory entries. In this paper, we represent the number of entries in the sparse directory as $R\times$ where R is the ratio of the number of sparse directory entries to the total number of blocks in the last-level private caches (e.g., L2 caches if each core has private L1 and L2 caches) aggregated over all processor cores. The traditional solution for keeping the volume of DEVs low is to have a sparse directory that is at least $1\times$ large; usually it is much larger to avoid worst-case conflicts because the sparse directory associativity is far lower than the impractical aggregate associativity of all the private last-level core caches. As a result, the sparse directory can easily consume several megabytes of on-chip area for a high-end server CMP.

In this paper, we propose novel techniques that, by design, *guarantee* complete freedom from DEVs irrespective of the sparse directory size, thereby enabling an unbounded sparse directory. Complete elimination of DEVs can bring multiple different benefits as discussed in the following.

A. Motivation: Benefits of Eliminating DEVs

1) Interconnect Traffic, Core Cache Misses, Performance: Elimination of DEVs can improve performance if the DEVs are predominantly live; live DEVs increase core cache misses and interconnect traffic. The performance improvement arising from elimination of DEVs is likely to be significant for small directory sizes. In the following, we first empirically establish that a $1\times$ sparse directory performs close to a system with an unbounded sparse directory for our workload set and system configuration.[1] This result allows us to use the $1\times$ sparse directory as a reasonable baseline throughout this paper and helps us avoid overstating any results. Next, we show that the performance degrades gradually with decreasing sparse directory size underscoring the performance-criticality of DEVs.

Figures 2 and 3 quantify the savings in interconnect traffic (total bytes communicated), core cache misses, and execution cycles for multi-programmed and multi-threaded workloads respectively when going from a $1\times$ sparse directory to an unlimited-capacity sparse directory. All results are collected on an 8-core CMP model with a non-inclusive cache hierarchy having an 8 MB 16-way shared LLC and per-core private 256 KB 8-way L2 cache and 32 KB 8-way data and instruction L1 caches. Each eight-way multi-programmed workload consists of eight copies of a SPEC CPU 2017 application (rate mode or homogeneous multi-programming) shown on the x-axis in Figure 2. Figure 3 shows the results for ten PARSEC applications [3], and the average numbers for the PARSEC, SPLASH2X (derived from SPLASH-2 [42] and distributed with PARSEC), SPEC OMP, and FFTW [13] applications. From Figure 2 we see that the average speedup is under 1%, although 10% interconnect traffic and 15% core cache misses are saved on average. To explain this, on top of each group of bars, we show the core cache misses saved per kilo instructions. These savings are too small (except 3.2 in xalancbmk) to affect any noticeable performance improvement (xalancbmk speeds up by 4%). Figure 3 shows that a $1\times$ sparse directory is adequate for the PARSEC, SPLASH2X, SPEC OMP, FFTW applications. The 4% performance loss in freqmine arises from forwarded requests to owner cores when using an unlimited-capacity directory. These requests were getting served from the LLC when using a $1\times$ sparse directory because the dirty blocks were retrieved from the owner cores as DEVs due to directory entry eviction.

The data in Figures 2 and 3 establish that our baseline $1\times$ directory performs close to the unbounded directory. Figure 4, however, shows that the performance declines gradually compared to the $1\times$ directory as the sparse directory is sized $\frac{1}{2}\times$, $\frac{1}{8}\times$, and $\frac{1}{32}\times$, thereby making the performance-criticality of the DEVs prominently visible. A solution that can eliminate DEVs must create the illusion of an unbounded virtual sparse directory. An important question that we seek to answer is whether such a solution can lower the requirement on the physical size of a dedicated on-chip sparse directory structure.

2) Isolation of Core Caches from Directory Evictions: Recent research [43] has cast shadow on the traditional directory-based CMP systems by demonstrating that DEVs can be exploited to launch Prime+Probe attacks [23], [28] under a threat

[1] All sparse directory organizations considered in this study are eight-way set associative (Table I of Section IV).

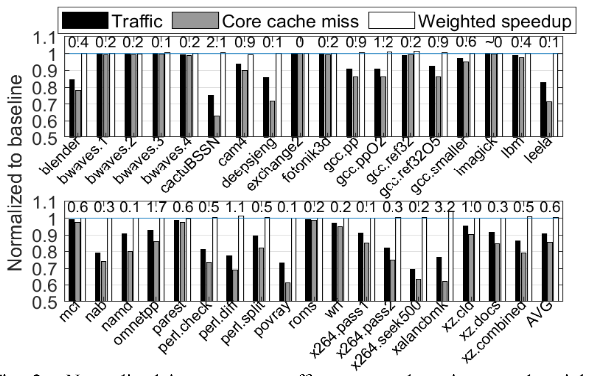


Fig. 2. Normalized interconnect traffic, core cache misses, and weighted speedup of eight-way rate (homogeneous) multi-programmed workloads.

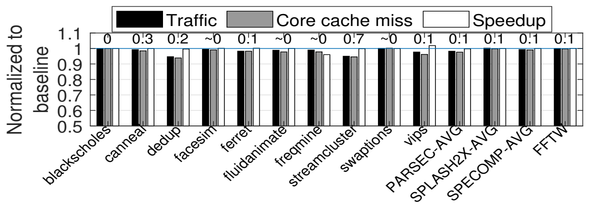


Fig. 3. Normalized interconnect traffic, core cache misses, and speedup of multi-threaded applications.

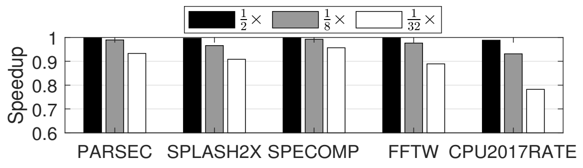


Fig. 4. Performance impact of sparse directory size.

model wherein a victim process and an active attacker process are scheduled within the same CMP sharing the sparse directory. Such attacks can reveal information (such as part of the physical address [19]) about the victim's accesses. The Secure Directory (SecDir) proposed to mitigate this attack divides the sparse directory into n private partitions and a shared partition for an n-core CMP [44]. A newly allocated directory entry starts its life in the shared partition. An entry E evicted from the shared partition gets allocated in the private partitions of the cores that are caching the block being tracked by E. Thus, a cross-core conflict in the shared partition can no longer directly generate a DEV, but can induce self-conflicts within a private partition due to migration of an entry from the shared partition to the private partition. An eviction from the private partition of core C due to self-conflicts within the private partition generates an invalidation to C leading to a DEV. This solution does avoid direct generation of DEVs arising from cross-core directory conflicts, but cross-core directory conflicts can indirectly generate DEVs by inducing self-conflicts within a core's directory entries. Clearly, a solution that can eliminate DEVs altogether could have offered complete isolation between the core caches and the directory evictions.

In summary, DEVs degrade performance, inflate interconnect traffic, and can be exploited to leak information through side-channels. The central contribution of this paper is the zero directory eviction victim (ZeroDEV) protocol that, by design, guarantees freedom from DEVs and offers a practically unbounded sparse directory interface to the core caches, which never receive any invalidation arising from directory entry eviction (Section III). In this paper, we squarely focus on the performance aspects of ZeroDEV and leave a thorough evaluation of the security aspects to future work. Our proposal, evaluated on inclusive and two popular non-inclusive cache hierarchy designs, performs within 1-2% of the baseline that uses a $1\times$ sparse directory. Importantly, due to elimination of DEVs and use of the LLC for caching directory entries, our proposal maintains this performance level with a significantly smaller sparse directory or no sparse directory structure at all (Sections IV and V).

II. Related Work

The volume of DEVs is closely related to the size and organization of the sparse directory. As a result, the large body of research on optimization of the space invested to the sparse directory has also effectively restricted the volume of DEVs [2], [7], [9], [10], [12], [32], [36], [37], [45], [50]–[54]. None of these proposals, however, guarantee freedom from DEVs. Ours is the first proposal to offer a DEV-free protocol for on-chip directory-based coherence. The two-level directory architecture proposed in a prior study for cc-NUMA multiprocessors uses additional space in the home memory to store a second-level directory to back up entries evicted from the per-node first-level directory cache, thereby eliminating node-level DEVs [1]. Our proposal, in contrast, focuses on eliminating DEVs within a CMP and engineers a set of novel run-time techniques to dynamically synthesize a multi-level caching hierarchy of sparse directory entries spread across the LLC and home memory. Our solution for eliminating DEVs related to intra-socket directory entry evictions does not require any additional storage. In-Cache Coherence Information (ICCI) tracking is the first proposal to store coherence tracking information in the LLC blocks [14], but this proposal requires request forwarding for all shared blocks increasing the critical path latency of reads to these blocks. Also, this proposal generates DEVs when an LLC block holding coherence tracking information is evicted. The Tiny Directory proposal addresses the critical path-related shortcoming of ICCI by incorporating a custom-designed tiny directory to track a critical subset of the read-shared blocks [37]. This proposal, however, cannot guarantee freedom from DEVs as replacement of a tiny directory entry generates invalidations just like a traditional directory entry eviction.

Three different categories of designs have emerged over time that have eliminated the sparse directory altogether or simplified coherence protocol/directory design. The first category of designs relies on timestamp-based leases for privately cached blocks and employs self-invalidation on lease expiry [25], [26], [34], [46]–[49]. These systems generate DEVs for privately cached exclusive/modified blocks when their copies are evicted from the LLC. The timestamp-based coherence idea has also been explored for GPUs [29], [38], [40]. The second category of designs takes help of operating system (OS)-controlled page mapping on private core caches and uses remote accesses to other cores' private caches to avoid replication of data in the private caches [11]. Thread migration across cores can also be used to enable access to other cores' private caches. As a result, run-time techniques that dynamically select between thread migration across cores and remote accesses to other cores' private caches have also been explored based on the observed affinity toward data elements [35]. These proposals do not generate DEVs, but require custom OS support. The third category of proposals relies on data-race-free software or fully labeled programs for identifying the acquire/release boundaries so that the privately cached copies of the blocks can be self-invalidated at appropriate program points [4], [8], [20], [30], [39]. Among these, VIPS-M [30], DeNovo [4], DeNovoND [39], and Dir_1-SISD [8] do not have DEVs, but require changes to the application software as well as OS kernels for correct self-invalidation. In contrast, our proposal eliminates DEVs while confining all design modifications to the uncore hardware and retains all benefits of a traditional directory-based cache coherence protocol; it can seamlessly run unmodified application binaries on top of stock operating system kernels.

As already discussed, a recent study has pointed out that DEVs can be exploited to launch timing-based side-channel attacks [43]. The defense mechanism, SecDir [44], avoids generation of DEVs arising directly from cross-core directory conflicts. In contrast, our ZeroDEV protocol eliminates DEVs altogether.

III. Design of the ZeroDEV Protocol

In this section, we discuss the details of the ZeroDEV proposal for designing a DEV-free system. We first summarize the baseline cache hierarchy architecture (Section III-A). Next, we discuss the overview of our main idea and present a data-driven analysis that motivates our approach (Section III-B). Sections III-C and III-D discuss the two central mechanisms of our proposal. Sections III-E and III-F discuss application of ZeroDEV to different LLC designs.

A. Baseline Cache Hierarchy

The baseline CMP has private L1 and L2 caches per core. The LLC (L3 cache) is banked and shared among all cores. A demand fill from main memory is always allocated in the LLC as well as in the L2 and L1 caches of the requesting core. An eviction from the LLC does not generate any invalidation to the core caches, thereby making the LLC non-inclusive of the L2 and L1 caches. A slice of the sparse directory resides alongside each LLC bank. The directory slice is responsible for tracking all privately cached copies of the blocks mapped to that LLC bank. However, the organization of the LLC bank and the sparse directory slice can be completely different. On receiving a request from a core, the LLC bank and the adjacent sparse directory slice are looked up in parallel. The coherence actions are decided based on the state of the retrieved sparse directory entry. A write-invalidate MESI cache coherence protocol keeps the private caches coherent. All evictions from the private cache hierarchy are notified to the sparse directory to keep the directory contents up-to-date and avoid unnecessary future invalidations [24]; the eviction notices for clean blocks (in E or S state) do not carry any data. To accelerate code sharing, a code block is always filled in the private caches in S state. A request forwarded to an owner core caching the requested block in M or E state is responded directly to the requester by the owner core making the critical path of such requests three-hop long [15], [22]; the owner core also sends a "busy clear" message to the home sparse directory slice to clear the busy/pending state of the corresponding directory entry.

B. Overview of the ZeroDEV Protocol

The ZeroDEV protocol consists of two mechanisms. The first mechanism utilizes the LLC space for caching directory entries that cannot be accommodated in the sparse directory. The challenge in this mechanism is to manage the increased LLC pressure so that the performance is not affected. Since this mechanism lengthens the life of a directory entry significantly, we expect most directory entries allocated in the LLC to get freed while in the LLC. However, the entries tracking hot blocks may survive their residency in the LLC and get evicted eventually. The second mechanism avoids generation of invalidations to the private caches at the time of such an eviction. In this mechanism, we exploit the important observation that a copy of the block (say, B) that the evicted directory entry was tracking is available in the private cache hierarchy of at least one of the cores. Therefore, the evicted directory entry can overwrite B in the main memory and store itself in the place of B not requiring any additional space.

Caching directory entries in the LLC is likely to have performance implications if not done judiciously. So, before embarking on the ZeroDEV design, we seek answers to two important questions: (i) what is the projected increase in LLC pressure due to directory caching? (ii) what is the estimated performance loss due to increased LLC pressure? Figure 5 answers the first question by summarizing the number of additional directory entries required in the unlimited-capacity directory compared to the baseline $1\times$ directory. This number is shown as a percentage of the number of LLC blocks assuming that one directory entry will be spilled in one full LLC block. Within each application suite, the left bar shows this percentage for the application requiring the maximum number of entries, while the right bar shows the average of the maximum counts across all applications of the suite. Overall, the maximum occupancy is around 12%, while the average is at most 10%. We note that 12% LLC occupancy corresponds to less than two ways of the baseline 16-way LLC.

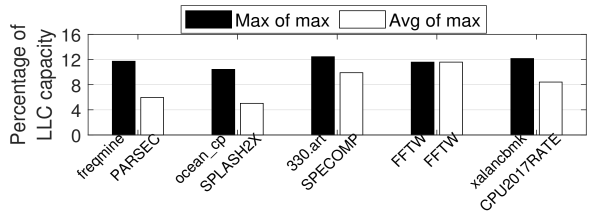


Fig. 5. Projected LLC occupancy of spilled directory entries.

Figure 6 offers an approximate answer to the second question by showing the performance as the associativity of all LLC sets is gradually reduced. The speedup numbers are normalized to the baseline 16-way LLC. On top of each bar in an application suite (except FFTW, which is a single-application suite), we show the speedup of the application that suffers from maximum slowdown within the suite. When only two ways are taken away from the LLC, the average performance loss is at most 3% (PARSEC). Within each suite, the following applications suffer most when two LLC ways are taken away: vips in PARSEC (14% loss in performance), lu_ncb in SPLASH2X (9% loss), 330.art in SPEC OMP (6% loss), and gcc.ppO2 in SPEC CPU 2017 rate (5% loss). An interesting related question is whether all directory entries can be housed in the LLC, thereby ridding the CMP of the sparse directory structure altogether. The number of entries in a $1\times$ sparse directory corresponds to 25% of the number of LLC blocks (arising from a 4:1 capacity ratio between the LLC and the private L2 caches). Figure 6 shows that when four ways (which is 25% of the LLC) are taken away from the baseline LLC, the average performance loss (see the 12-way group) is about 4% for PARSEC and 2% for SPALSH2X, SPEC OMP, and SPEC CPU 2017 rate. However, the maximum slowdown within each group is significant: 22% for vips in PARSEC, 17% for lu_ncb in SPLASH2X, 14% for 330.art in SPEC OMP, and 9% for gcc.ppO2 in SPEC CPU 2017 rate. In summary, these results show that although the average performance loss due to directory caching in the LLC is not significant, the naïve scheme of spilling directory entries into the LLC can lead to large worst-case performance losses. There is a need for designing smarter schemes of directory entry caching in the LLC.

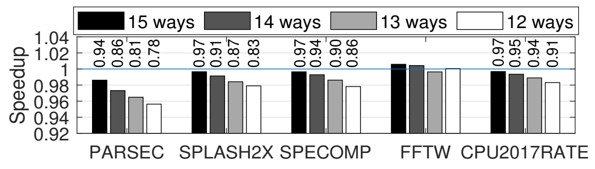


Fig. 6. Performance with reduced LLC associativity.

C. Caching Directory Entries in LLC

In this section, we present three different policies for accommodating directory entries in the LLC. We begin our discussion with the naïve policy of spilling directory entries into the LLC. We assume that each LLC block has two state bits, namely valid (V) and dirty (D). The following three states are used in the baseline: invalid (V=0, D=0), clean valid (V=1, D=0), and dirty valid (V=1, D=1).

1) SpillAll Policy: In the *SpillAll* policy, a valid entry evicted from the sparse directory is allocated in the LLC. Figure 7 shows a directory entry E tracking the privately cached copies of a block $B4$ resident in the LLC. When E is evicted from the sparse directory, it is allocated in the same set as $B4$ by replacing $B2$. To keep the design simple, we allocate a full LLC block to a spilled directory entry. Also, we let a spilled directory entry exercise the same set index function as the regular LLC blocks. To distinguish between a block and its spilled directory entry (e.g., $B4$ and E) residing in the same LLC set, we mark the spilled directory entry with state (V=0, D=1). The tag match lines coming out of the tag comparators (or the tag CAM) are ANDed with $\sim$V&D to generate a new set of match lines used to access the directory entry from the data array, while the original match lines ANDed with V are used to access the actual data block. If a tag lookup reveals two matching tags in the target set, the directory entry is looked up first in the data array. While the decoding of the directory entry progresses, the actual data block is accessed.

This policy does not require any change to the coherence protocol, but suffers from two important shortcomings: (i) spilled directory entries increase LLC pressure, and (ii) the requests to blocks in S state see an additional data array lookup latency on the critical path if the directory entry is spilled in the LLC. We

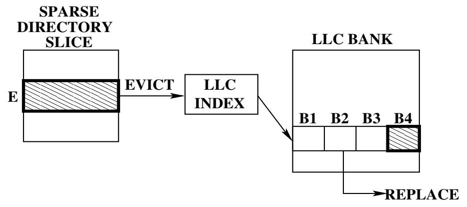


Fig. 7. SpillAll policy.

discuss ways to address these shortcomings in the next policy. We note that the updates to the spilled directory entries need additional LLC writes, but there is ample free LLC read/write bandwidth.

2) FusePrivateSpillShared Policy: To bring down the LLC pressure of the SpillAll policy, we make one important observation: to respond to a request R for an LLC block B that has coherence state M/E, the contents of B are not required. This is because R has to be forwarded to the core caching the latest copy of B in M/E state. Therefore, some part of an LLC block B that is in the coherence state M/E (i.e., temporarily private) can be used for storing its directory entry; this won't have any impact on the critical path of the next request to B. Such directory entries will be referred to as fused directory entries. The directory entries of the other blocks (i.e., blocks in S state) are spilled into the LLC space as in the SpillAll policy. Therefore, the percentage of directory entries that track shared blocks can offer an estimate of the increase in LLC pressure for this *FusePrivateSpillShared* policy. On average, this percentage is usually small: for PARSEC 10%, for SPLASH2x 19%, for SPEC OMP 0.5%, for FFTW nearly zero, and for SPEC CPU 2017 rate 9% (arising from code blocks being cached in shared state). While this is only an empirical estimate, even in general, the footprint of actively shared blocks (i.e., copies present in private caches) is maximized when the sharing degree is two and this footprint corresponds to only half of the directory entries in a $1\times$ sparse directory. This directory entry population corresponds to only two LLC ways of a 16-way LLC (due to 4:1 capacity ratio between the LLC and the private L2 caches). Overall, this policy is expected to offer significant relief to the LLC pressure. Figure 8 depicts the operations of this policy. A sparse directory entry E is tracking the privately cached copies of a block $B4$ resident in the LLC. When E gets evicted, it is spilled into the LLC set containing $B4$ if the coherence state of the block is S; otherwise E is fused with $B4$ by overwriting several bits of $B4$.

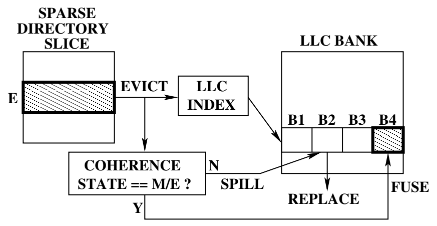


Fig. 8. FusePrivateSpillShared policy.

Both fused and spilled entries use the state (V=0, D=1) to distinguish them from regular LLC blocks. Figure 9 shows the formats of the spilled and fused entries. The least significant bit (say, b_0) of an LLC block in state (V=0, D=1) indicates whether it is a spilled or a fused entry. The rest of the bits (e.g., 511 bits in a 64-byte LLC block) in a spilled entry store the directory entry. For a fused entry, the encoding of the remaining bits (b_1 onward) of the LLC block is as follows (assuming N cores): LLC block dirty (b_1), directory state busy (b_2), owner encoding ($b_3, \ldots, b_{2+\lceil \log_2 N \rceil}$), and the remaining portion of the LLC data block. When a core evicts a block in E state, it needs to send back the least significant $3 + \lceil \log_2 N \rceil$ bits to the LLC along with the eviction notice message so that the fused LLC block can be reconstructed and can be returned to (V=1, D=1) or (V=1, D=0) state depending on whether the LLC block is dirty or not. These extra bits in the eviction notice message of a block in the E state introduce negligible interconnect traffic overheads compared to the baseline. The M state evictions generate full-block writeback messages, as in the baseline.

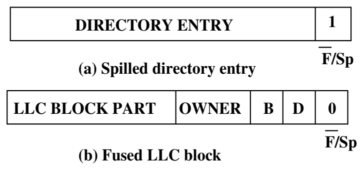


Fig. 9. Format of spilled and fused entries. F=fused, Sp=spilled, D=dirty, B=busy.

Next, we turn to understand the impact of this policy on the critical path of requests coming from the core caches. To ensure that the critical path of a read request is not lengthened, this policy maintains the invariants that (i) if a directory entry is fused in the LLC, its coherence state must be M/E, and (ii) if a directory entry is spilled in the LLC, its coherence state must be S. Recall that the lengthened critical path scenario of read requests for the SpillAll policy was related to the case where an LLC lookup (done in parallel with the sparse directory lookup) returns two tag matches in the target LLC set. Now, two tag matches in an LLC set necessarily implies that one tag (in state V=1) corresponds to an LLC block B and the other corresponds to the block's spilled directory entry E_B. In this case, the aforementioned invariants imply that B must be in the S state. Therefore, if the request is a read, B can be read out first and sent as response to the requester even before E_B is read out. Subsequently, E_B is read out and updated off the critical path. Thus, in this case, the baseline critical path latency of reads is preserved by this policy. For upgrade requests, only E_B is read out and the count of expected invalidation acknowledgments is included in the dataless response. For read-exclusive requests, both B and E_B are read out one by one and the count of expected invalidation acknowledgments is included in the response along with the data.

If an LLC lookup returns just one tag match, the corresponding block is read out, as in the baseline. The (V, D) states along with the least significant bit of the block are examined (only if V=0 and D=1) and the appropriate coherence action is initiated. Note that if the state of the block turns out to be (V=0, D=1), the request must be forwarded to an owner core (having the block in the M/E state) or to a sharer core, which will directly respond to the requester. This is similar to the baseline case where a request hits in the sparse directory, but misses in the LLC.

To maintain the two aforementioned invariants, whenever a block transitions from the S state to the M state and if its directory entry is spilled in the LLC, the directory entry is fused with the block and the spilled entry is freed. On the other hand, when a block transitions from the M/E state to the S state and if its directory entry is fused, the directory entry is spilled into the same set by invoking the LLC replacement policy. Also, at this time, the block is reconstructed by having the owner core send back the least significant $3 + \lceil \log_2 N \rceil$ bits of the block to the LLC along with the busy/pending clear message. These bits introduce negligible interconnect traffic overhead. In summary, the FusePrivateSpillShared policy effectively addresses both the shortcomings of the SpillAll policy.

3) FuseAll Policy: To complete the design space of caching directory entries in the LLC, we consider a policy where an entry evicted from the sparse directory is fused with the corresponding LLC block provided the block is present in the LLC irrespective of the coherence state of the block; if the block is not present in the LLC, the directory entry is spilled into the LLC. This *FuseAll* policy is inspired by the In-Cache Coherence Information (ICCI) tracking proposal, which does not have a sparse directory and uses parts of an LLC block to store the block's directory entry [14]. We appropriately modify this proposal by augmenting it with a sparse directory to derive the FuseAll policy. Figure 10 shows the operations of the FuseAll policy. When a sparse directory entry E is evicted, it is fused with the LLC block $B4$, the privately cached copies of which the entry E is tracking.

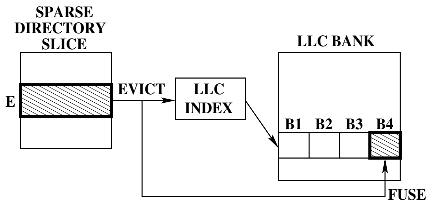


Fig. 10. FuseAll policy.

The FuseAll policy requires three different formats for representing a directory entry accommodated in the LLC. Figure 11 shows these formats. The corrupted portion of a fused block encodes the following pieces of information starting from the least significant side: fused/spilled (bit b_0), LLC block dirty (b_1), busy state of directory entry (b_2), M/E or S state of directory entry (b_3)[2], owner encoding if state is M/E ($b_4, \ldots, b_{3+\lceil \log_2 N \rceil}$) or sharer vector if state is S ($b_4, \ldots, b_{3+N}$), and the remaining portion of the LLC block (N is assumed to be the number of cores). Overall, depending on the state of the block, a fused entry has $4 + \lceil \log_2 N \rceil$ or $4 + N$ bits corrupted. In this policy, on receiving the eviction notice from the last sharer core of a fused block, a special acknowledgment message is sent to this sharer to retrieve the least significant $4 + N$ bits of the block so that the fused LLC block can be reconstructed and returned to (V=1, D=1) or (V=1, D=0) state. A sharer needs to preserve an evicted block in its eviction buffer until the eviction is acknowledged by the home LLC bank. This additional interconnect traffic overhead is negligible because this is introduced only once during the entire sharing life time of some of the shared blocks. The eviction notices for the E state blocks carry the least significant $4 + \lceil \log_2 N \rceil$ bits, as already discussed in the previous policy.

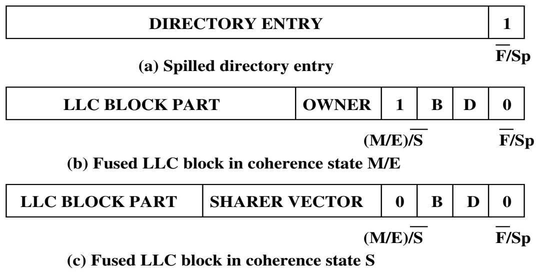


Fig. 11. Format of spilled and fused entries in FuseAll policy. F=fused, Sp=spilled, D=dirty, B=busy.

The FuseAll policy nearly nullifies the additional LLC pressure arising from directory entry caching. However, it introduces a significant performance problem for read requests to the shared blocks. All requests to the shared blocks now need to be forwarded to a sharer elected by the coherence controller because the corrupted LLC block cannot provide the requested data (except for upgrade requests which do not need a data response). For read-exclusive requests, this forwarded message can be combined with the invalidation request to the elected sharer, thereby keeping the critical path same as the baseline. However, for read requests, the critical path gets strictly increased to three hops compared to two hops in the baseline.

Figure 12 summarizes the design space of directory caching policies considered in this study as a function of LLC space overhead and increase in the critical path of reads to shared blocks, the directory entries of which have been accommodated in the LLC. The SpillAll policy has the maximum LLC space overhead while the read critical path gets lengthened by the LLC data array lookup latency. The FusePrivateSpillShared (FPSS) policy has only LLC overhead and no critical path overhead for reads. The FuseAll policy has small LLC overhead due to a small number of spills for the directory entries the corresponding blocks of which have already been evicted from the LLC, but this policy lengthens the critical path of reads to shared blocks by one extra hop.

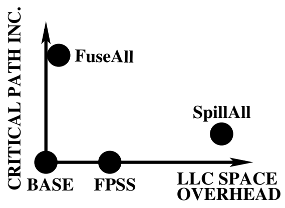


Fig. 12. Design space for directory entry caching in the LLC.

4) Replacement-disabled Sparse Directory: The ZeroDEV protocol has an option of having sparse directories that do not have any replacement policy, thereby simplifying the design. In such a design, a new directory entry first looks for an invalid way in the target sparse directory set and if none found, it gets accommodated in the LLC in fused or spilled form according to the directory entry caching policy of the LLC. Note that a valid entry residing in the sparse directory will eventually get freed (i.e., become invalid) when the block it is tracking becomes non-shared/unowned. Disabling replacement from the

[2] The directory cannot distinguish between M and E states, as in the baseline [22].

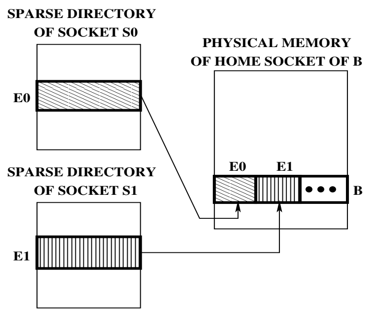
sparse directory in ZeroDEV is a strictly better design option because in such a design, a directory entry victimizes only one sparse directory entry or LLC block (depending on where it gets allocated) during its entire life time. On the other hand, in a replacement-enabled sparse directory, a directory entry E can victimize one directory entry (when E first gets allocated in the sparse directory) and later one LLC block (when E is evicted from the directory and moved to the LLC), thereby causing disturbance in both structures. When our ZeroDEV protocol is equipped with a sparse directory, we always assume that the sparse directory is replacement-disabled and hence, simpler to design.

D. Directory Entry Eviction from LLC

In this section, we discuss the second important component of our ZeroDEV proposal, namely handling eviction of fused or spilled directory entries from the LLC without generating any invalidations to the private caches. When a valid fused or spilled directory entry corresponding to a block B is evicted from the LLC, it is clear that at least one core C is caching a copy of B. Therefore, overwriting B in the physical memory to store the evicted directory entry does not lead to any data loss; B can be recovered from core C. ZeroDEV implements this idea. We assume the existence of a socket-level coherence directory for maintaining inter-socket coherence using a home-based MESI protocol similar to the intra-socket protocol. Each socket-level directory entry has three stable coherence states, namely M/E, S, and I, encoded using two state bits. The unused fourth state is used by ZeroDEV to encode whether the home memory block is in a corrupted state due to storage of a directory entry. To keep the socket-level directory entries up-to-date, a socket on evicting its last copy of a block notifies the home coherence controller.

To understand the basic scheme of the proposal, let us consider a scenario depicted in Figure 13, where two sockets $S0$ and $S1$ are caching a block B and the corresponding intra-socket sparse directory entries are $E0$ and $E1$, respectively. When $E0$ or $E1$ is evicted from the corresponding socket's sparse directory, it is moved to the LLC of the respective socket; when it is evicted from the LLC, it is housed within B. ZeroDEV partitions the home memory block B into fixed segments and reserves each segment for housing a directory entry from a socket. Therefore, $E0$ or $E1$ is housed in the portion of B that is reserved for the corresponding socket. If each socket has N cores, a valid intra-socket sparse directory entry in a stable state would require $N + 1$ bits of storage (N bits for sharing-vector/owner, one bit for two coherence states M/E and S). Assuming 64-byte memory blocks, this arrangement can support up to $\lfloor \frac{512}{N+1} \rfloor$ sockets. For scaling beyond these socket counts for a given N, one can explore imprecise or compressed storage of the evicted intra-socket directory entries that are housed in the physical memory blocks. For example, a hybrid of limited-pointer and coarse-vector formats can dynamically choose between precise and imprecise representations depending on the sharer count of an evicted sparse directory entry. Our study in this paper maintains the full-map bitvector representation.

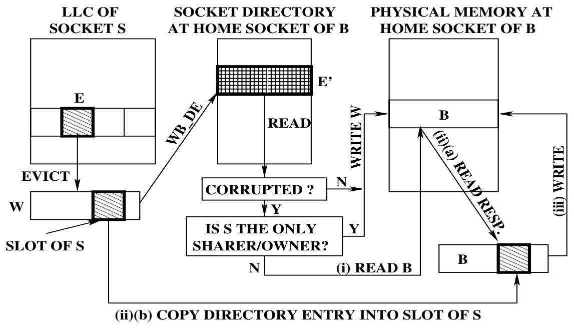
Figure 14 depicts the flow of a directory entry (DE) write-back (WB_DE) triggered when a fused or spilled directory entry E corresponding to a block B is evicted from the LLC of some socket S. The entry E is held up in a buffer inside socket S if



SPARSE DIRECTORY OF SOCKET S0
E0
PHYSICAL MEMORY OF HOME SOCKET OF B
SPARSE DIRECTORY OF SOCKET S1
E0
E1
B
E1

Fig. 13. Housing live intra-socket directory entries in physical memory block.

it is in a transient state. Once E returns to a stable state (M/E or S), the LLC controller prepares a 64-byte block W with E positioned in the segment reserved for the source socket S. W is sent with a directory entry writeback message (opcode WB_DE) to the home socket H of B. If the state of the home socket directory entry E' of B is not corrupted or corrupted with socket S marked as the only sharer/owner, W is written to home memory. The socket directory entry state switches to corrupted leaving the sharer vector unchanged (socket S remains marked as a sharer/owner). On the other hand, if the state of E' is corrupted with at least one more socket other than S marked as a sharer, the coherence controller at H executes the following steps: (i) reads out B from home memory, (ii) extracts the evicted entry E from W and copies it into the appropriate position within B, and (iii) writes B back to home memory.



LLC OF SOCKET S
SOCKET DIRECTORY AT HOME SOCKET OF B
PHYSICAL MEMORY AT HOME SOCKET OF B
E
E'
WB_DE
READ
WRITE W
B
(ii)(a) READ RESP.
(iii) WRITE
EVICT
W
SLOT OF S
CORRUPTED ?
N
Y
IS S THE ONLY SHARER/OWNER?
Y
N
(i) READ B
B
(ii)(b) COPY DIRECTORY ENTRY INTO SLOT OF S

Fig. 14. Flow of operations on a directory entry eviction from LLC.

Overall, a directory entry eviction from the LLC is expensive because each eviction requires a DRAM write and in a multi-socket system, some of the evictions may require DRAM reads. In the following, we discuss simple extensions to the baseline LLC replacement policy to reduce the volume of directory entry eviction. We also present a few new extensions to the inter-socket coherence protocol.

1) Extensions to LLC Replacement Policies: We discuss two simple modifications to the baseline LRU policy to reduce the eviction volume of fused and spilled entries. The first modification uses the observation that whenever an LLC block B is accessed, the corresponding directory entry E_B is also accessed. In such situations, we update the LRU position of B first and then the LRU position of spilled E_B (if any) putting E_B in the MRU position. Since both B and spilled E_B belong to the same LLC set, this update rule guarantees that B would be evicted before spilled E_B gets evicted. This policy further extends the LLC residency of the spilled directory entries. We will refer to this policy as *spill protect LRU* or spLRU.

The spLRU policy fails to offer any additional protection to the fused entries, the eviction of which would also require a DRAM write to back up the fused directory entry even if the state of the fused block is not dirty. We propose an extension to the baseline LRU policy that evicts the ordinary LLC block (in state V=1) closest to the LRU position first before evicting any spilled or fused entries in an LLC set. This policy, referred to as *dataLRU*, evicts all ordinary data/code blocks in a set first before evicting any spilled or fused entries. We note that these policy extensions are simple and require minor modifications to the existing logic. Moreover, the replacement policy is not on the critical path.

2) Handling Uncore Hits Within a Socket: A core cache miss request arriving at the home LLC bank may encounter one of four possible scenarios: (i) the requested block is present in the LLC and the corresponding directory entry is found inside the socket (in the sparse directory or in the LLC), (ii) the requested block is not present in the LLC, but the corresponding directory entry is found inside the socket, (iii) the requested block is present in the LLC, but the corresponding directory entry is not found inside the socket, and (iv) the requested block is not present in the LLC and the corresponding directory entry is not found inside the socket. The first three cases constitute uncore hits (LLC block hit or directory entry hit) within a socket. The protocol actions for the first two cases are similar to the corresponding cases in the intra-socket protocol. The third case requires considering two possible sub-cases: (iiia) there is no sharer/owner of the block in the socket, or (iiib) there are sharers/owner of the block in the socket, but the directory entry has been evicted and written back to home memory. Distinguishing between these two sub-cases is expensive, as it requires querying the inter-socket coherence directory. Fortunately, our LLC replacement policy extensions guarantee that sub-case (iiib) cannot arise because an LLC block would be evicted before (or together with) its spilled (or fused) directory entry. Therefore, if the requested block is present in the LLC, but the corresponding directory entry is not found within the socket, we can conclude that the block has no sharer/owner within the socket (same as sub-case (iiia)). The protocol actions for handling this sub-case are same as those for case (iii) in the baseline. That leaves us with case (iv). This case constitutes a socket miss. In the following, we discuss the protocol extensions needed to handle a socket miss.

3) Extensions for Handling Socket Misses: Figure 15 shows the salient steps after a miss request originating from socket S reaches home socket H. If the socket-level directory entry at H indicates a non-corrupted state (i.e., M/E, S, I), the protocol actions are same as baseline (step ② in Figure 15). If the socket-level directory entry indicates a corrupted state with socket S being a sharer or the owner, the protocol actions at home are same as baseline (step ③). In this case, the memory block is read out and sent to the requesting socket S using a special message type indicating that the returned block is corrupted. Socket S incurs one cycle additional delay to extract its intra-socket sparse directory entry from the returned block. The rest of the actions are similar to the baseline intra-socket protocol case where the directory entry for the requested block is present within the socket, but the requested block is not present in the LLC.

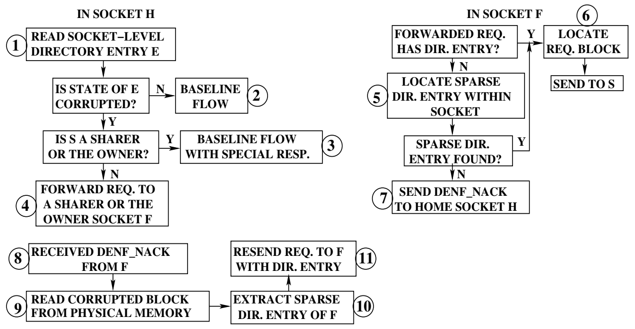

IN SOCKET H
① READ SOCKET–LEVEL DIRECTORY ENTRY E
IS STATE OF E CORRUPTED?
N
BASELINE FLOW ②
Y
IS S A SHARER OR THE OWNER?
Y
BASELINE FLOW WITH SPECIAL RESP. ③
N
④ FORWARD REQ. TO A SHARER OR THE OWNER SOCKET F
IN SOCKET F
⑥
FORWARDED REQ. HAS DIR. ENTRY?
Y
LOCATE REQ. BLOCK
SEND TO S
N
⑤ LOCATE SPARSE DIR. ENTRY WITHIN SOCKET
SPARSE DIR. ENTRY FOUND?
Y
N
⑦ SEND DENF_NACK TO HOME SOCKET H
⑧ RECEIVED DENF_NACK FROM F
⑨ READ CORRUPTED BLOCK FROM PHYSICAL MEMORY
EXTRACT SPARSE DIR. ENTRY OF F ⑩
RESEND REQ. TO F WITH DIR. ENTRY ⑪

Fig. 15. Flow of an LLC miss from socket S. H is the home socket and F is a socket where H may have to forward the request.

If the socket directory entry indicates a corrupted state and the requesting socket S is not a sharer or the owner, the only way to get the requested block is to forward the request to a socket F that is a sharer or the owner (step ④). F is selected by H from the socket-level directory entry. On receiving the forwarded request, F first tries to locate the intra-socket sparse directory entry of the requested block within its socket (step ⑤). If F can find the directory entry, it retrieves the requested block from a sharer core or the owner core within its socket and sends it to S (step ⑥). On the other hand, if F fails to find a directory entry, there are two possible reasons: (i) F still has copies of the requested block, but has evicted and written back the intra-socket directory entry, or (ii) F has evicted its last copy of the requested block and sent the block to H to find out if this is the last system-wide copy of the block. In the first case, F sends a "directory entry not found negative acknowledgment" (DENF_NACK) back to the home socket H (step ⑦). In the second case, a copy of the block would be found in the eviction buffer of LLC in F waiting for an acknowledgment from H. This copy is used by F to send a response to S (this corner case is not shown in Figure 15 for brevity). When H receives a DENF_NACK message (step ⑧), it extracts the directory entry of F from the block in home memory and forwards the original request again to F along with the directory entry using a different message type (steps ⑨ to ⑪). A different message type prompts F to use the directory entry in the forwarded message and F concludes the request (step ⑥).

Clearly, in the case discussed above (i.e., socket-level directory entry indicating a corrupted state of the home memory block with the requester not being a sharer or the owner), the critical path of LLC misses can get lengthened significantly. Thanks to our LLC replacement policy extensions, this case is encountered rarely; less than 0.5% of DRAM writes arise from directory entry eviction indicating a tiny population of corrupted blocks in home memory. Further, a very small fraction (less than 0.05%) of LLC read misses access corrupted blocks. The majority of accesses to the corrupted blocks arise from core cache evictions that remove a sharer or the owner from the sparse directory entry segment belonging to the originating socket. We discuss this next.

4) Handling Evictions from Core Caches: Having discussed how ZeroDEV handles all the cases of core cache miss requests, we turn to understand how core cache evictions are handled. Figure 16 depicts the major steps involved in handling an eviction coming from a core C within socket S. The eviction is fielded by the home LLC bank in socket S (step ①). If the eviction fails to locate a sparse directory entry within the socket, but it is a writeback message carrying a full cache block, the inference is that the evicting core must be the system-wide owner of the block (in M state). The baseline protocol flow for writeback to home socket is executed (step ②).

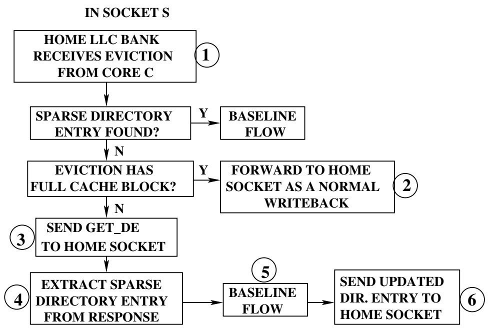

IN SOCKET S
HOME LLC BANK RECEIVES EVICTION FROM CORE C
1
SPARSE DIRECTORY ENTRY FOUND?
Y
BASELINE FLOW
N
EVICTION HAS FULL CACHE BLOCK?
Y
FORWARD TO HOME SOCKET AS A NORMAL WRITEBACK
2
N
3
SEND GET_DE TO HOME SOCKET
4
EXTRACT SPARSE DIRECTORY ENTRY FROM RESPONSE
5
BASELINE FLOW
SEND UPDATED DIR. ENTRY TO HOME SOCKET
6

Fig. 16. Flow of a core cache eviction request that cannot find the sparse directory entry within the requesting socket S.

If the sparse directory entry is not found within socket S and the eviction message does not carry the full cache block, an access to the intra-socket directory entry is required to understand the situation. Therefore, a directory entry (DE) read request (GET_DE) is sent to the home socket (step ③). In response, the home socket sends the (necessarily) corrupted block to S. The requesting socket extracts its intra-socket sparse directory entry from the response (step ④). Now, it executes the usual intra-socket protocol flow for handling the eviction e.g., removes the evicting core from the the directory entry (step ⑤). Finally, it sends the updated directory entry back to the home socket for writing back (step ⑥). During execution of steps ③ to ⑥, any core cache request to the same block is negatively acknowledged.

As in the baseline, if the last sharer core or the owner core (in E state) within S has evicted the block and the block is not present in the LLC, an eviction notice is sent to the home socket to update the socket-level directory entry. Additionally, in the ZeroDEV protocol, if the socket-level directory entry indicates that the state of the home memory block is corrupted and this is the system-wide last copy of the block, the block is retrieved from the evicting core and sent to the home socket to overwrite the corrupted memory block.

5) Handling Socket-level Directory Evictions: In the following discussion, we assume that a socket-level directory cache implements the socket-level directory. This can be either an SRAM cache [21] or a much larger DRAM cache [5], [18]. An eviction from this directory cache can generate DEVs throughout the system jeopardizing the isolation between core caches and directory evictions. We discuss two solutions to handle this issue. The first solution is to back up the socket-level directory in home memory. This solution has been used widely in large-scale distributed shared memory multiprocessors [21], [22]. On a directory cache miss, the home memory is looked up for the directory entry. For a quad-socket system, the DRAM overhead of this solution is only 1.2% assuming 64-byte memory blocks. For a 32-socket system, the overhead is 6.6%, still within reasonable limits. Since this solution is simple and has low DRAM overhead for small socket-count systems, our multi-socket evaluation on four sockets uses this solution for both baseline and our proposal.

Our second solution makes the DRAM overhead a constant. This solution extends ZeroDEV to socket-level directory entries as well. This is achieved by reserving a partition within each memory block for housing an evicted socket-level directory entry. In an M-socket system, this partition would require $M+2$ bits (M bits for sharer vector and two state bits). Thus, a memory block would have $M+1$ partitions: M partitions for intra-socket directory entries and one for the socket-level directory entry. Given N cores per socket and 64-byte memory blocks, the upper bound on M is found by solving $512 \geq M(N+1)+(M+2)$ i.e., $M \leq \lfloor \frac{510}{N+2} \rfloor$. When an evicted socket-level directory entry is housed in the home memory block, a separate *DirEvict* bit per memory block records this status. Thus, instead of backing up a full directory entry in home memory, just a bit is maintained per block bringing down the DRAM overhead to 0.2%. This overhead does not depend on the socket-count. On a directory cache miss, the *DirEvict* bit of the block is consulted. If this bit is set, the socket-level directory entry is extracted from the home memory block (which is anyway looked up in parallel with the directory cache access). The protocol actions are same as what have been discussed for a block in the corrupted state. In general, requests to corrupted blocks need to be forwarded to a sharer socket or the owner socket to get the data unless the requester itself is marked as a sharer or the owner, in which case step ③ from Figure 15 is invoked. To improve performance, the array of the *DirEvict* bits can be cached; an 8 KB cache can capture the *DirEvict* bits of 64K blocks and cover 4 MB of home memory footprint. Also, the volume of corrupted shared blocks should be minimized to reduce the impact on the critical path of reads to shared blocks. This can be achieved by associating a higher replacement priority to the owned blocks (in M/E state) in the directory cache.

6) Discussion: Protocol Complexity: We have discussed in reasonable detail the life of an intra-socket directory entry as it moves from the on-chip sparse directory to the LLC and then to the home memory. While most of the extensions incorporated on top of the baseline protocol are quite routine, one specific protocol path is significantly more complex than others; this path is invoked when handling a socket miss to a corrupted block with the requester not marked as a sharer or the owner (see Section III-D3). Although this case has a lot of resemblance with the baseline protocol case when a request needs to be forwarded to another socket, the situation is complicated by the possibility of a racing sparse directory entry eviction in the forwarded socket as discussed in Section III-D3. Generating the rule-sets governing this protocol case and the related invariants requires careful consideration. Overall, the ZeroDEV protocol extensions require a few new message types and one new stable state (fused/spilled/corrupted) in the LLC and the socket-level directory entry. It also requires small additional storage in main memory to handle socket-level directory entry evictions. In exchange, ZeroDEV rids the entire system of invalidations arising from directory entry evictions.

E. LLC with Exclusive Private Data

Our baseline design always fills a block in the LLC when fetching it from the main memory, thereby significantly improving the chances of fusing directory entries with the corresponding LLC blocks. However, an important class of non-inclusive non-exclusive LLC design does not fill a block into the LLC when fetching it from the main memory (similar to AMD Magny Cours [6]). In such designs, the fetched block is filled only in the private cache hierarchy of the requesting core in E or M state. A block gets allocated in the LLC if (i) the block is evicted from the private cache hierarchy of the owner core, or (ii) the block gets shared; allocating a shared block in the LLC can accelerate future sharing by making the subsequent shared-read requests conclude in two hops. Additionally, whenever a block transitions to M or E state (meaning that it becomes temporarily private), it is de-allocated from the LLC. In general, temporarily private data blocks (i.e., in M or E state) are not allocated in the LLC and can be found exclusively in private caches, thereby avoiding replication of such blocks in the LLC and improving cache space utilization. We will refer to such a design as exclusive private data (EPD) LLC. As in the baseline non-inclusive LLC, an LLC replacement does not invalidate the copies (if any) of the evicted block from the core caches.

In an EPD LLC, the privately owned blocks in M/E state are not present in the LLC. So, directory entry fusion for these blocks is not possible in the LLC. The directory entries of these blocks must be spilled into the LLC when they are evicted from the sparse directory. Since these blocks occupy a major portion of the application working set, the ZeroDEV proposal needs assistance from a sparse directory to keep the directory caching pressure in the EPD LLC within reasonable limits.

F. Inclusive LLC

A cache hierarchy with an inclusive LLC also suffers from forced core cache invalidations due to sparse directory entry eviction. The ZeroDEV proposal applies seamlessly to an inclusive LLC and can help eliminate all DEVs. An important observation is that an inclusive LLC does not experience any directory entry eviction from the LLC. This is because the ZeroDEV LLC replacement policies (see Section III-D) victimize the code/data blocks before (or together with) their spilled (or fused) directory entries. To maintain the inclusion property, the privately cached copies of these blocks must also be invalidated, thereby freeing the corresponding directory entries even before they are evicted. Therefore, the baseline inter-socket coherence protocol does not need any change in an inclusive LLC.

IV. Simulation Environment

We use the Multi2Sim simulation infrastructure [41] to evaluate our proposal. Table I lists the parameters of a simulated socket. We use CACTI [17] to determine the lookup latency of the cache arrays shown in Table I. It is important to note that the LLC tag and data lookup latency numbers mentioned in this table are only for the array lookup. The round-trip latency for LLC lookup includes, in addition to these array lookup latency numbers, the latency to traverse the interconnect, the lookup latency at the inner-level caches, and the waiting time at several interface queues up and down the cache hierarchy including the port queues in the interconnect switches. The multi-socket evaluations are done for four sockets with an inter-socket routing delay of 20 ns. To evaluate the scalability of our proposal, we also model a 128-core single-socket system having a 32 MB 16-way shared LLC, per-core 128 KB 8-way L2 cache and 32 KB 8-way L1 caches, and eight single-channel DDR3-2133 controllers.

TABLE I
Baseline simulation environment (one socket)

CPU core (eight in number, dynamically scheduled, x86, 4 GHz)
224-entry ROB, 128-entry LSQ, iL1 & dL1 cache: 32 KB/8-way, L2 cache: 256 KB/8-way, all caches: 64-byte block, LRU policy
Shared LLC, sparse directory, interconnect
LLC: 8 MB/16-way/8 banks/LRU/3-cycle tag lookup/ 4-cycle data access/64-byte block. Sparse directory: 8-way, 1-bit NRU. Interconnect: 2D mesh, 1-cycle routing delay, 1-cycle link latency.
Main memory (modeled using DRAMSim2 [31])
Two single-channel DDR3-2133 controllers, 64-bit channel, BL=8, two ranks per channel, x8 DRAM devices, eight banks, 1 KB row buffer per bank, latency parameters: 14-14-14-35

Table II shows the multi-threaded applications used in our evaluation. The PARSEC, SPLASH2X, SPEC OMPM 2001, and FFTW applications are executed for the entire region of interest (ROI). The throughput-oriented server workloads are evaluated on 128 cores by replaying a trace of instructions collected using PIN.

TABLE II
Multi-threaded applications

PARSEC (input sizes within parentheses)
blackscholes (large), canneal (large), dedup (medium), facesim (large), ferret (large), fluidanimate (large), freqmine (large), swaptions (large) streamcluster (medium), vips (large)
SPLASH2X (inputs within parentheses)
fft (16M points), lu_cb (2048×2048 matrix), radix (64M keys), lu_ncb (2048×2048 matrix), ocean_cp (1026×1026 grid), radiosity (1.5e-2 BFepsilon), raytrace (anti-aliasing with 2 subpixels, balls4.env), water_nsquared (medium size from SPLASH2X inputs), water_spatial (medium size from SPLASH2X inputs)
SPEC OMPM 2001 (inputs within parentheses)
312.swim (ref with 3 iters), 314.mgrid (ref with 1 charge, 1 iter), 316.applu (train with 6 pseudo-timesteps), 320.equake (ref with ARCHduration 0.01), 324.apsi (train with 1 timestep), 330.art (train 2)
FFTW (inputs within parentheses)
FFTW (256×256×256 points)
SERVER (inputs, configuration, simulation length within parentheses)
SPEC jbb (82 warehouses, single JVM instance, 6 billion instructions), Apache HTTP server (SPEC Web-Banking (B)/Ecommerce (E)/ Support (S), 128 simultaneous sessions, worker thread model, mod_php, 5 billion instructions), MySQL TPC-C (10 GB DB, 2 GB buffer, 100 warehouses, 100 clients, 500 transactions), MySQL TPC-E (10 GB DB, 2 GB buffer, 100 clients, 5 billion instructions), MySQL TPC-H (2 GB DB, 1 GB buffer, 100 clients, zero think time, even mix of Q6, Q8, Q11, Q13, Q16, Q20, 5 billion instructions)

We prepare 36 homogeneous (rate) and 36 heterogeneous 8-way multi-programmed workloads using the SPEC CPU 2017 applications. These applications were shown in Figure 2 (all application-input pairs when using the ref inputs). We ensure that each application has equal representation in the heterogeneous workload mixes, thereby avoiding any bias. Each application in a multi-programmed workload retires a representative segment of 500M dynamic instructions picked using the SimPoint tool [33]. Early finishing applications continue running until each application in the workload retires the representative instruction set.

V. Simulation Results

We begin our evaluation of the ZeroDEV proposal by selecting the directory entry caching policy (from among SpillAll, FusePrivateSpillShared, and FuseAll) and the LLC replacement policy (from among spLRU and dataLRU). Next, we evaluate the sensitivity of ZeroDEV to different system parameters and compare its performance with related proposals.

Selection of Directory Entry Caching Policy. Figure 17 compares the speedup of ZeroDEV working with three policies relative to the baseline. To make a robust selection, we maximize the directory footprint in the LLC by completely disabling the sparse directory of ZeroDEV. ZeroDEV executes the dataLRU LLC replacement policy in this study. On top of each bar, we show the minimum speedup of any application within a suite (except FFTW, which has only one application). As expected, SpillAll is the worst policy. While the average speedup numbers of FusePrivateSpillShared (FPSS) and FuseAll are close, the minimum speedup numbers clearly show that FusePrivateSpillShared is a superior policy. The FuseAll policy reduces the number of LLC misses, but significantly lengthens the critical path of read requests to shared blocks. Additionally, we note that the savings in the interconnect traffic for SpillAll and FPSS are similar to what we observed in Figures 2 and 3 due to reduction in core cache misses. However, due to the extra forwarded read requests, the interconnect traffic in FuseAll increases by about 9% on average compared to FPSS for the multi-threaded workloads. Further, we observed that the performance gap between the FusePrivateSpillShared and FuseAll policies increases gradually with increasing core-count, as the performance penalty of the lengthened critical paths in the FuseAll policy is significantly more in larger systems. In the rest of this section, we will operate ZeroDEV with the FusePrivateSpillShared policy.

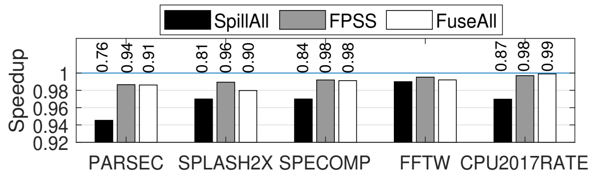


Fig. 17. Comparison between SpillAll, FusePrivateSpillShared (FPSS), and FuseAll policies on an 8-core single-socket system.

Selection of LLC Replacement Policy. Figure 18 shows the speedup achieved by ZeroDEV (without a sparse directory) operating with the spLRU or dataLRU policy for 8 MB LLC (sp8MB, data8MB bars) and 4 MB LLC (sp4MB, data4MB bars) in an 8-core system. For reference, the baseline 4 MB LLC performance executing LRU replacement policy is shown in Base4MB. The 4 MB LLC results help us clearly see the difference between the spLRU and dataLRU policies because any inefficiency would be significantly magnified in a capacity-constrained LLC. All results are normalized to baseline 8 MB LLC. Across the board, the dataLRU policy is higher performing. The spLRU policy fails to offer protection to the fused directory entries and increases the DRAM traffic for reading and updating such directory entries. In the rest of this section, ZeroDEV will use the dataLRU policy.

Impact of Sparse Directory Size on ZeroDEV. Having fixed the directory caching and LLC replacement policies for ZeroDEV, we show its detailed performance in Figures 19, 20, and 21 for an 8-core system having a shared 8 MB LLC. In

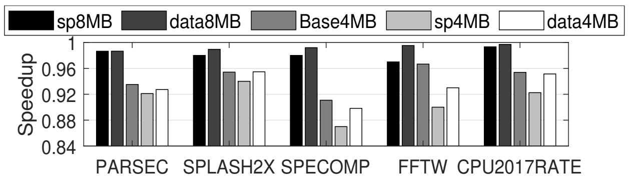


Fig. 18. Comparison between spLRU and dataLRU.

these results, ZeroDEV is evaluated with three different sparse directory configurations, namely $1\times$, $\frac{1}{8}\times$, and no directory. All results are normalized to the baseline having a $1\times$ sparse directory. Across all the suites, we observe that the performance of ZeroDEV is nearly invariant of the sparse directory size. Most importantly, ZeroDEV without any sparse directory performs within a percentage of the baseline for all the suites, on average (see the GEOMEAN bars). The primary reason for this remarkable result is the judicious use of the LLC space for caching the directory entries. We find that for ZeroDEV operating without a sparse directory, the average DRAM read traffic increases by at most 2% for any of the application suites (the primary reason for small performance loss in ZeroDEV), while the increase in the average DRAM write traffic is less than 0.5% relative to the baseline. For the PARSEC suite (Figure 19), freqmine has the largest slowdown (expected result and explained in Figure 3 of Section I). For the SPLASH2X, SPEC OMP, and FFTW suites (Figure 20), lu_ncb, raytrace, water_nsquared, and 330.art suffer from 1-4% slowdown. For the SPEC CPU 2017 rate workloads (Figure 21), cam4 suffers from the largest slowdown of 2%. Across all suites, ZeroDEV delivers performance within a percentage of the baseline, on average, for all three directory configurations.

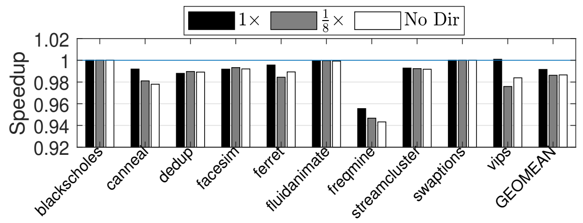


Fig. 19. Performance of ZeroDEV on the PARSEC suite.

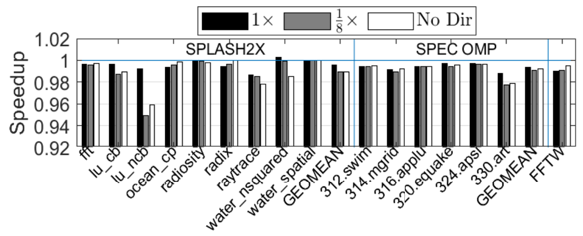


Fig. 20. Performance of ZeroDEV on SPLASH2X, SPEC OMP, FFTW.

Sensitivity to LLC Capacity. Figure 22 evaluates ZeroDEV for 4 MB and 16 MB shared LLCs (both 16 ways). All results are normalized to the baseline with 8 MB LLC. For 16 MB LLC capacity, ZeroDEV operating without a sparse directory performs within a percentage of the 16 MB baseline (Base16MB). For 4 MB LLC capacity, ZeroDEV needs some assistance from a sparse directory (results shown with a $\frac{1}{4}\times$ directory) in the case

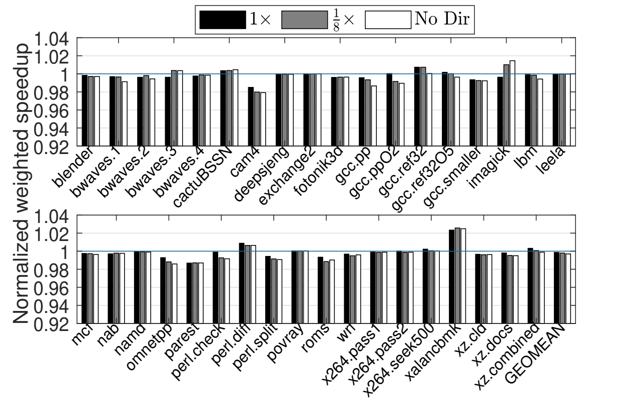


Fig. 21. Performance of ZeroDEV on the SPEC CPU 2017 rate workloads.

of a few applications (e.g., FFTW) to keep the LLC pressure within reasonable limits and perform within a percentage of the 4 MB baseline (Base4MB).

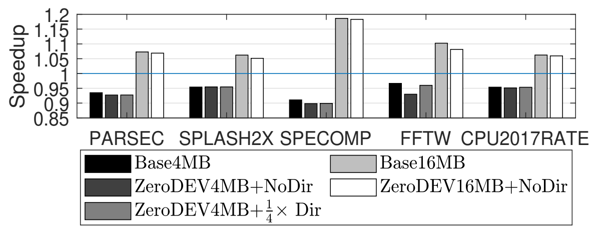


Fig. 22. Performance with 4 MB and 16 MB shared LLC.

Heterogeneous Multi-programmed Workloads. Figure 23 evaluates ZeroDEV operating with three directory configurations (1×, $\frac{1}{8}$×, and no directory) for the heterogeneous multi-programmed workloads running on 8 cores. The individual workload slowdown is at most 2%, while, on average, all three configurations of ZeroDEV perform within a percentage of the baseline which has a 1× sparse directory.

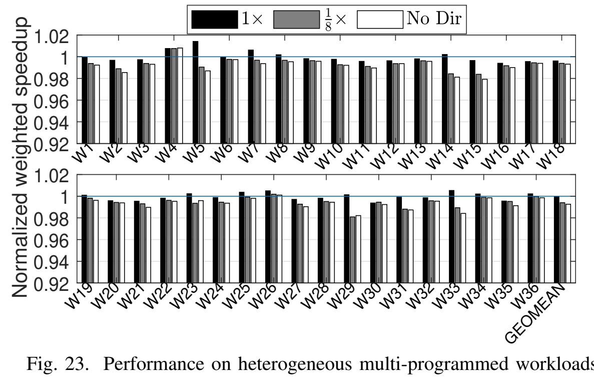


Fig. 23. Performance on heterogeneous multi-programmed workloads.

Server Workloads. Figure 24 evaluates ZeroDEV operating with three directory configurations (1×, $\frac{1}{8}$×, and no directory) on the server workloads. This evaluation is done on a 128-core single-socket system with a 32 MB 16-way shared LLC. When ZeroDEV has no directory, the maximum slowdown is 1.4% (SPECWeb-S). Across all three configurations, the average performance is within a percentage of the baseline.

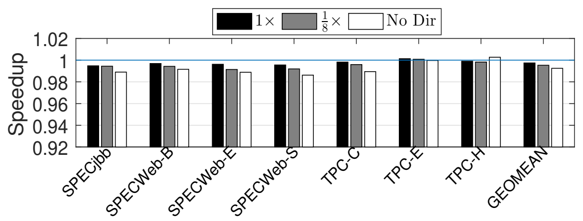


Fig. 24. Performance on server workloads (128-core single-socket).

Performance on EPD and Inclusive LLCs. Figure 25 evaluates ZeroDEV on exclusive private data (EPD) and inclusive LLCs of capacity 8 MB (for the server applications, the LLC capacity is 32 MB). All results are normalized to the baseline non-inclusive LLC running with a 1× sparse directory. CPU-RATE and CPU-HET groups respectively refer to the homogeneous and heterogeneous multi-programmed workloads. For each group of applications, the leftmost three bars show the baseline EPD LLC performance (BaseEPD) for three sparse directory sizes (1×, $\frac{1}{2}$×, $\frac{1}{8}$×); the next three bars show ZeroDEV performance on EPD LLC (ZeroDEVEPD) for three sparse directory configurations (no directory, $\frac{1}{2}$×, 1×); the rightmost two bars show the performance of baseline inclusive LLC (BaseIncl) with 1× sparse directory and ZeroDEV performance on top of inclusive LLC working without a sparse directory (ZeroDEVIncl+NoDir). Across the board, the baseline EPD LLC with 1× and $\frac{1}{2}$× sparse directories performs better than the baseline non-inclusive LLC having a 1× sparse directory. This performance advantage comes from better cache space utilization in the EPD LLC (see Section III-E). ZeroDEV with EPD LLC performs within 1-2% of the corresponding EPD LLC baseline when equipped with $\frac{1}{2}$× and 1× sparse directories. Interestingly, for several application groups, ZeroDEVEPD without a sparse directory outperforms the EPD LLC baseline having a $\frac{1}{8}$× sparse directory. This is because ZeroDEVEPD can use the LLC space for directory caching. Overall, ZeroDEVEPD maintains acceptable performance compared to the baseline EPD LLC design for all directory configurations. However, it is desirable to have a sparse directory with ZeroDEV when incorporated in an EPD LLC because directory entry fusion is not possible in the LLC (see Section III-E); due to excessive directory entry spilling in the LLC, ZeroDEVEPD+NoDir loses significant performance for some applications when compared to BaseEPD with 1× sparse directory (e.g., FFTW).

ZeroDEV implemented in an inclusive LLC with no sparse directory (ZeroDEVIncl+NoDir) performs within 1-2% of the baseline inclusive design (BaseIncl). Interestingly, we notice that ZeroDEV eliminates 95% of the forced invalidations from the core caches in the inclusive design. The remaining forced invalidations arise due to inclusion property of the LLC.

Comparison to Related Work. We compare the performance of ZeroDEV with two recent related proposals. The first one (Multi-grain Directory [50]) improves the sparse directory space investment, while the second one (SecDir [44]) addresses the problem of DEV-related side-channel attacks. In Figure 26, we compare our proposal with the Multi-grain Directory (MgD), which significantly reduces the overhead of tracking private blocks by investing just one directory entry to track a private region of size 1 KB. The leftmost three bars in each application

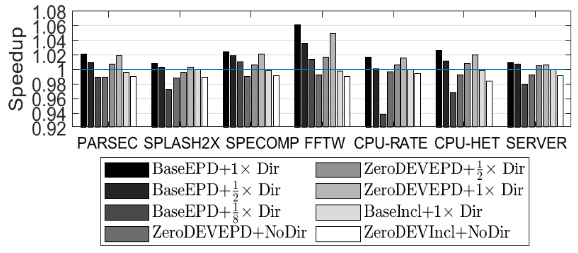


Fig. 25. Performance on exclusive private data (EPD) and inclusive LLCs.

group show the performance of MgD for $\frac{1}{8}\times$, $\frac{1}{16}\times$, and $\frac{1}{32}\times$ directory sizes. The next three bars show ZeroDEV performance for $1\times$, $\frac{1}{8}\times$, and no directory. All results are collected for the non-inclusive LLC configuration and normalized to baseline $1\times$ directory configuration. While MgD with a $\frac{1}{8}\times$ directory offers performance similar to the baseline $1\times$, its performance degrades gradually as the directory size is further reduced (we note that this performance is still much better than the baseline with identical directory sizes). As already discussed, the decline in average performance of ZeroDEV with shrinking directory size is within 1% across the board. Thus, the performance gap between ZeroDEV and MgD rapidly widens with shrinking sparse directory size.

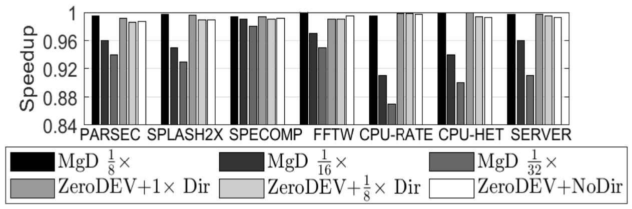


Fig. 26. Performance comparison with Multi-grain Directory.

Figure 27 compares the performance of SecDir and ZeroDEV. The SecDir proposal was introduced in Section I-A2. The left-most three bars in each application group show SecDir with $1\times$ directory, baseline with $\frac{1}{8}\times$ directory, and SecDir with $\frac{1}{8}\times$ directory. The next three bars show ZeroDEV performance for $1\times$, $\frac{1}{8}\times$, and no directory. For both SecDir sizes ($1\times$, $\frac{1}{8}\times$), the comparison is iso-storage meaning that the number bits devoted to the sparse directory of SecDir is nearly same as in the baseline. The overall number of directory entries in SecDir $R\times$ size is more than that in baseline $R\times$ size because an entry in the private partition of SecDir does not need to maintain the sharer list or owner information, thereby saving bits. In the $1\times$ directory configuration of SecDir for an 8-core system, each baseline directory slice having 512 sets and 8 ways is partitioned into eight private zones each having 32 sets and 7 ways and a shared zone having 512 sets and 5 ways; in the $\frac{1}{8}\times$ configuration, the number of sets in each partition is made one-eighth of what the $1\times$ configuration has keeping the associativity unchanged. For the 128-core system (the server group), in the $1\times$ configuration of SecDir, each baseline sparse directory slice having 256 sets and 8 ways is partitioned into 128 private zones each having 4 sets and 8 ways and a shared zone having 256 sets and 4 ways. In the $\frac{1}{8}\times$ configuration, each private partition is four-way fully associative and the shared partition has 32 sets and 4 ways.

In Figure 27, on top of the bars for SecDir+$1\times$ Dir, SecDir+$\frac{1}{8}\times$ Dir, and ZeroDev+NoDir, the minimum speedup numbers achieved by any application within a group are noted to understand the maximum slowdown observed in these designs. While SecDir loses performance with decreasing sparse directory size ($1\times$ to $\frac{1}{8}\times$) as the baseline also does, ZeroDEV remains mostly unaffected by the varying sparse directory size and performs within a percentage of the $1\times$ baseline. The minimum speedup figures for SecDir indicate large slowdown at $\frac{1}{8}\times$ directory size due to internal fragmentation in the private partitions. For the server group (evaluated on 128 cores), the internal fragmentation becomes so severe that the average performance loss relative to the baseline $\frac{1}{8}\times$ configuration is 11% while the maximum slowdown is 18%.

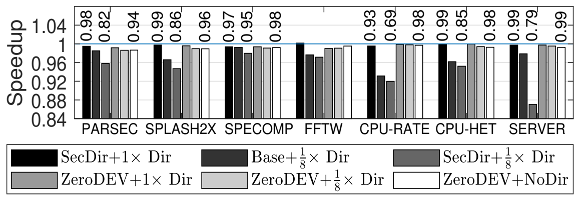


Fig. 27. Performance comparison with SecDir.

Energy Expense. ZeroDEV can save area and leakage energy by eliminating the sparse directory in inclusive and non-inclusive LLC designs. However, it also increases the LLC dynamic energy due to additional reads and writes to the directory entries accommodated in the LLC. Using CACTI we estimate that ZeroDEV running without a sparse directory can save about 9% energy, on average, in the sparse directory and the LLC taken together compared to the baseline running with a non-inclusive LLC of capacity 8 MB (32 MB for server applications) and a $1\times$ sparse directory.

Multi-socket Evaluation. We evaluate ZeroDEV on a four-socket system with each socket having eight cores and an 8 MB non-inclusive shared LLC. We use the PARSEC, SPLASH2X, SPEC OMP, FFTW, SPEC CPU 2017 rate and heterogeneous workloads in this evaluation. Each heterogeneous multi-programmed workload is scaled up to have a mix of 32 applications. Each homogeneous (rate) multi-programmed workload now has 32 copies of the same SPEC CPU 2017 application. Each multi-threaded application is executed with 32 threads. Across these groups of workloads, ZeroDEV operating without an intra-socket sparse directory performs, on average, within 1.6% of the baseline which has a $1\times$ sparse directory for intra-socket coherence.

VI. Summary

We have presented the ZeroDEV protocol, the first design to guarantee freedom from directory eviction victims within a CMP. A scheme for efficiently caching directory entries in the LLC and a mechanism for handling directory entry eviction from the LLC without generating invalidations to the core caches are at the center of the ZeroDEV design. The end-result is that the core caches enjoy the illusion of an unbounded directory and remain completely isolated from directory eviction. For a large set of multi-threaded and multi-programmed workloads, ZeroDEV performs within 1-2% of a well-provisioned traditional baseline, which delivers performance close to that of an unlimited-capacity sparse directory. Interestingly, for inclusive and a class of non-inclusive LLCs, ZeroDEV maintains its performance level without requiring an intra-socket directory. While ZeroDEV, by design, has isolated the core caches from directory entry evictions, a thorough study of the security aspects of ZeroDEV is an important future work.

Acknowledgment

The author would like to thank the anonymous reviewers for all the feedback.

References

[1] 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.

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

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

[4] 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.

[5] C-C. Chou, A. Jaleel, and M. K. Qureshi. CANDY: Enabling Coherent DRAM Caches for Multi-node Systems. In *MICRO* 2016.

[6] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. In *IEEE Micro*, January/February 2010.

[7] 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.

[8] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras. An Efficient, Self-Contained, On-Chip Directory: DIR$_1$-SISD. In *PACT* 2015.

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

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

[11] C. Fensch and M. Cintra. An OS-based Alternative to Full Hardware Coherence on Tiled CMPs. In *HPCA* 2008.

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

[13] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *ICASSP* 1998.

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

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

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

[17] 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/.

[18] C-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan. C3D: Mitigating the NUMA Bottleneck via Coherent DRAM Caches. In *MICRO* 2016.

[19] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE S&P* 2013.

[20] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. In *IEEE Micro*, September/October 2010.

[21] J. Kuskin, D. Ofelt, M. A. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. L. Hennessy. The Stanford FLASH Multiprocessor. In *ISCA* 1994.

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

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level Cache Side-Channel Attacks are Practical. In *IEEE S&P* 2015.

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

[25] S. L. Min and J. L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. In *IEEE TPDS*, January 1992.

[26] S. K. Nandy and R. Narayan. An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems. In *International Workshop on Parallel Processing*, December 1994.

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

[28] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference on Topics in Cryptology*, February 2006.

[29] X. Ren and M. Lis. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *HPCA* 2017.

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

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

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

[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS* 2002.

[34] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Davadas. Library Cache Coherence. *MIT CSAIL Technical Report MIT-CSAIL-TR-2011-027*, May 2011.

[35] K. S. Shim, M. Lis, O. Khan, and S. Devadas. The Execution Migration Machine: Directoryless Shared-Memory Architecture. In *IEEE Computer*, September 2015.

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

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

[38] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, T. M. Aamodt. Cache Coherence for GPU Architectures. In *HPCA* 2013.

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

[40] A. Tabbakh, X. Qian, and M. Annavaram. G-TSC: Timestamp Based Coherence for GPUs. In *HPCA* 2018.

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

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

[43] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *IEEE S&P* 2019.

[44] M. Yan, J-Y. Wen, C. W. Fletcher, and J. Torrellas. SecDir: A Secure Directory to Defeat Directory Side-channel Attacks. In *ISCA* 2019.

[45] 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.

[46] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures. In *ICS* 2016.

[47] X. Yu and S. Devadas. TARDIS: Timestamp based Coherence Algorithm for Distributed Shared Memory. In *PACT* 2015.

[48] X. Yu, H. Liu, E. Zou, and S. Devadas. Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models. In *PACT* 2016.

[49] X. Yuan, R. Melhem, and R. Gupta. A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence. In *ICPP* 1996.

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

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

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

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

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