SelectDirectory: A Selective Directory for Cache Coherence in Many-Core Architectures

Yuan Yao[1], Guanhua Wang[2], Zhiguo Ge[3], Tulika Mitra[2], Wenzhi Chen[1] and Naxin Zhang[3]
[1]College of Computer Science and Technology, Zhejiang University
[2]School of Computing, National University of Singapore
[3]Huawei International Pte. Ltd.
Email: yuanyao@zju.edu.cn, wangguan@comp.nus.edu.sg, ge.zhiguo@huawei.com,
tulika@comp.nus.edu.sg, chenwz@zju.edu.cn, naxin.zhang@huawei.com

***Abstract*—As we move into many-core era fueled by Moore's Law, it has become unprecedentedly challenging to provide the shared memory abstraction through directory-based cache coherence. The main difficulty is the high area and power overhead of the directory in tracking the presence of a memory block in all the private caches. Sparse directory offers relatively better design trade-offs by decoupling the coherence meta-data from the last-level cache (LLC); but still suffers from high area/power issues. In this work, we propose a compact directory design by exploiting the observation that a significant fraction of the memory blocks are temporarily exclusive in the cache hierarchy and hence only needs minimal sharer information. Inspired by this observation, we propose to further decouple the tag array from the coherence meta-data array in the sparse directory and allocate a sharer list only for the actively shared blocks. Experimental results reveal that our proposal, called SelectDirectory, can substantially save directory storage area and energy without sacrificing performance.**

I. Introduction

A considerable consensus has been reached that cache coherence will continue to be employed in future large-scale systems [6][13]. With the rapid increase in the number of cores on chip, the scalability of a coherence protocol is highly challenging — maintaining coherence across hundreds or thousands of cores will be unprecedentedly difficult. Although directory coherence protocols offer a relatively practical approach, there is growing concern that simply applying the directory coherence to many-core architecture will face serious power and area issues.

There are several baseline directory architectures used for on-chip coherence, such as duplicate tag directory, in-cache directory and sparse directory. *Duplicate Tag* directory [1] is area-efficient but it becomes less attractive in many-core era due to the prohibitive energy consumption in its highly associative structure. In-Cache directories [13] encode the per-core coherence states in every LLC entry. Though it is relatively more energy-efficient, the redundant storage for the uncached blocks makes the design area-inefficient. *Sparse* directory [3] designs use a directory cache to flexibly store directory entries, decoupled from the last level cache (LLC). The sparse directory with low-associativity offers energy and area efficiency. Unfortunately, set conflicts can occur frequently due to the low associativity in the sparse directory. Sparse directories also require *back invalidation* of the private cache blocks if the corresponding sparse directory entry is evicted, inevitably degrading system performance. Over-provisioning alleviates this problem by having more directory entries than the aggregated number of private cache entries. For example, a sparse directory with twice the number of private cache entries (denoted as Sparse $2\times$) can remarkably diminish the invalidation rate but still does not eliminate the effect. Sacrificing area for reduced directory-induced invalidation can be a mediocre design choice for small-scale systems, but the unnecessarily over-sized directory scales poorly with the growing on-chip core counts.

To make the directory coherence more scalable, extensive works have been proposed on efficient sharing pattern representation [3][7][20] and hierarchical directories [11][20]. Other approaches like Cuckoo directory [12] and SCD [20] make more effective use of the directory by using a multi-hashing based insert and replacement policy, thus reducing the set conflicts. Recently, SCT [2] and MGD [9] employ multi-granular entries to compact the directory storage.

In this work, we take a different approach that exploits the observation that a significant fraction of the memory blocks stored in the on-chip cache hierarchy are *temporarily private* [2][9][10]. These blocks only need minimal sharer information. We leverage this observation by proposing SelectDirectory that decouples the tag array and the data (i.e. coherence meta-data) array of the sparse directory, and allocates data entries only for the actively shared blocks. Experimental results reveal that SelectDirectory can reduce the data array size by 8x with identical performance to the baseline Sparse $2\times$. We also synergistically combine SelectDirectory with MGD [9] coherence directory for further directory size savings.

Decoupling cache tag and data has been proposed in the context of LLC management [8][14], purposefully improving LLC utilization, but without taking coherence tracking reduction into account. In our proposal, we seek an efficient sparse directory to reduce coherence tracking overheads. The decoupled directory structure allows less data entries than tag entries, which well matches our requirements for selective data allocation in the directory.

II. Motivation

Actively shared blocks are simultaneously cached by more than one private cache. In a sparse directory, we define a directory entry as an *actively shared entry* if it is tracking multiple sharers. In this section, we first acquire the theoretical upper bound of the number of actively shared directory entries for a sparse directory. Then by exploiting characteristics germane to directory coherence, we take a statistical approach to further identifying the amount of actively shared entries.

Theoretical Upper Bound for Actively Shared Directory Entries Every block cached in the private caches is tracked by a directory entry. At a given time, the total number of blocks

978-3-9815370-4-8/DATE15/©2015 EDAA

cached in the private caches is equal to number of tracked sharers in the directory. Assuming there are N cores, let d_n be the number of directory entries that track n sharers, and $\mathcal{M}$ be the aggregated number of private cache entries, we have:

$$\sum_{i=1}^{N} d_i \times i = \mu\mathcal{M}$$

Where μ is the occupation ratio of private caches. We define E_{active_shared} to be the number of actively shared directory entries, so we have:

$$E_{active\_shared} = \sum_{i=2}^{N} d_i$$

We define ρ to refer to the ratio of actively shared directory entries to the aggregated number of private cache entries:

$$\rho = E_{active\_shared}/\mathcal{M}$$

Clearly, ρ maximizes when $\mu = 1$ and all the blocks in the private caches are actively shared. In particular, ρ reaches its theoretical limit when all the blocks have exactly two sharers. In such case, the number of actively shared directory entries will be $\mathcal{M}/2$, where ρ is 50%:

$$MAX(E_{active\_shared}) = \mathcal{M}/2, \quad MAX(\rho) = 50\%$$

Probability Distribution of ρ We proceed to use realistic workloads to characterize and identify the probability distribution of ρ. In order to eliminate the perturbation from set conflict, we set the directory size large enough to ensure no eviction could occur. For each run, the workload creates the same number of threads as the core count and runs to completion. More simulation settings can be found in Section IV.

To get the probability distribution of ρ throughout the execution of applications, we periodically take a snapshot and record ρ for all 16 workloads. For each workload, we set the time interval to uniformly obtain 500 snapshots (or samples). Figure 1 plots the individual and overall cumulative probabilities of ρ for all 16 workloads. The y axis is the probability of $\rho <= x$, We can see that for most workloads, $P(\rho <= 15\%) > 0.95$.

The bold curve in the figure is the overall cumulative probability of all 8000 (500 x 16) samples. The overall cumulative probability gradually approaches to 1 where $\rho <= 20\%$. It reveals that at any given point during the entire application execution, ρ rarely exceeds 20%.

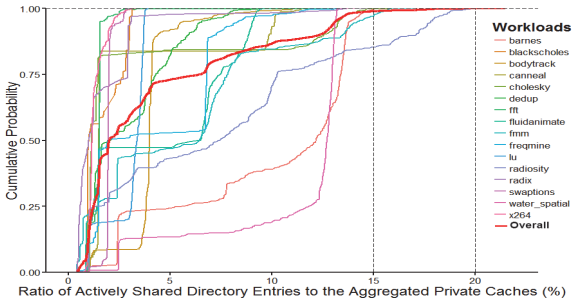


Fig. 1: Individual and overall cumulative probabilities of ρ for 16 workloads.

In summary, we conclude that the number of actively shared directory entries has a theoretical limit of $\mathcal{M}/2$. But workload characterization reveals that at any given point during application execution, it almost always remains under $\mathcal{M}/5$.

III. Design of SelectDirectory

The previous section motivates that at any given point in time, a minor fraction of directory entries are actively shared. This section describes a practical and efficient design (SelectDirectory) that takes advantage of this observation. We first present the organization of SelectDirectory, followed by its operation description and effects on the coherence protocol, finally the comparison with closest work.

A. Organization

Similar to associative caches, a conventional sparse directory arranges the tag and state information in a tag array and the tracked sharers in a data (or meta-data) array. For the identities of the sharers, a bit vector is usually used where one bit represents a corresponding core. We exploit the aforementioned sharing pattern to design SelectDirectory that allocates an entry in the data array only when the block becomes actively shared. In addition, when the block transitions back to temporarily private state or gets evicted in the data array, the data entry is deallocated. Temporarily private blocks only have tag entires allocated. Thus, the data array size can be greatly reduced.

Figure 2 shows the structure of SelectDirectory and the formats of tag and data entries. By decoupling the tag array and the data array, SelectDirectory breaks the one-to-one mapping of tag and data; thus the data array can have fewer number of entries than the tag array. As shown in Figure 2, a forward pointer and a reverse pointer are employed for associating the two arrays [14][8]. The tag array the index bits to identify the set whereas the data array uses a subset (least significant bits) to identify the set. Specifically, the forward pointer links the tag entry to one data entry in the set, and a reverse pointer in the data array indicates the associated tag position in the tag array. Each tag entry is also extended with an owner pointer, which is necessary for temporarily private blocks to track its exclusive owner.

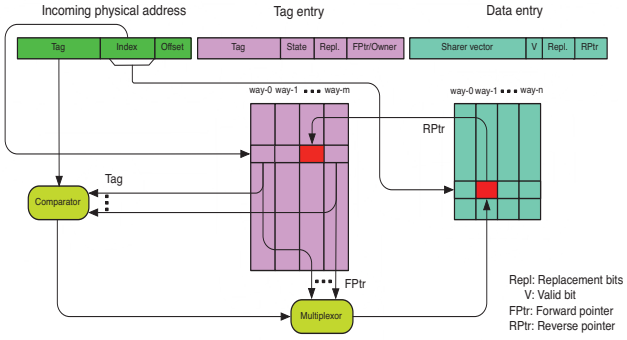


Fig. 2: Structure and entry formats of SelectDirectory.

As the position of a data entry is encoded in the forward pointer, a lookup in the data array does not require any associative search. So increasing the associativity of the data array does not have an impact on power consumption.

However, when increasing associativity of the data array, we need a bigger forward pointer in the tag array. Take the example of a 32-core CMP, we use $SD(t\times, d\times)$ to refer to the numbers of tag entries and data entries relative to the

aggregated number of all private cache entries, where t and d are the ratios of the tag array and the data array respectively. For a fully associative data array, every tag entry will have a $\log_2(data\_entris)$ bit forward pointer.

On the other hand, note that either the forward pointer or the owner pointer is *active* at any point in time. For a temporarily private block, the forward pointer bits are unused as it only has a tag entry allocated. While for an actively shared block, the owner pointer is redundant as it has been encoded in the sharer vector. Thus, to minimize storage overhead of the design, the forward pointer and the owner pointer can share the same storage in the tag array. The directory controller interprets the bits differently according to the block's sharing status. We now describe detailed operations of SelectDirectory.

B. Operations

Lookup and Data Allocation Upon a miss in the tag array, instead of conventionally allocating both tag and sharer vector, SelectDirectory only allocates tag and keeps the owner in the tag entry. Upon a hit in the tag array, several situations can happen depending on the block states and the request type.

If the block is temporarily private, a read request will make the block become actively shared. Then SelectDirectory allocates a data entry and encodes the owner and the requester in the sharer vector. Simultaneously, it sets the forward pointer in the tag array. We call it an *upgrade* action. A write request will keep the block as temporarily private, but the owner will be changed, and the request is forwarded to the old owner.

If the block is actively shared, the forward pointer indicates the way number of the corresponding data entry in the data array. The operations on a read request are conventional. However, for a write request, the block becomes temporarily private again, which opens up opportunities for creating space in the data array. Therefore, in parallel to sending invalidation messages to the sharers, SelectDirectory deallocates the entry in the data array and overwrite the forward pointer with the new owner. We denote this action as *downgrade*.

Replacement and Private Cache Eviction The replacement can happen both in the tag array and the data array. In this work, we use the LRU replacement policy for the two arrays. A tag replacement will evict both the tag and the data entries if it is linked to a data entry. For a data array eviction, SelectDirectory will perform another type of *downgrade*. It keeps the tag entry in the tag array; in addition, it picks one sharer and invalidates the others. The forward pointer in the tag entry is then covered by the selected owner. This optimization reduces the number of blocks to be invalidated, and keeping the tag and one sharer alive also improves latency for future access to the block.

On receiving eviction notifications from private caches, a block with only a tag entry will be marked invalid. For a block with tag and data entries, one possible situation is that only one sharer is left after the private cache eviction. In this case, we do not proactively downgrade the data entry to the tag array, because the data entry is likely soon to be evicted and in turn downgraded by another entry. When receiving the last sharer's private cache eviction notification, both the tag entry and the data entry will be reclaimed by SelectDirectory.

C. Effects on the Coherence Protocol

In SelectDirectory, a block is allowed to have no data entry allocated. SelectDirectory requires the sharing status of the block to perform appropriate operations. Fortunately, we find the coherence states of the directory controller is adequate to obtain the information, which does not introduce any extra overhead. In a conventional MESI coherence protocol, the sparse directory controller has three stable states [11]: M, S, I. E and M are both represented by M in the directory. A block in M state is potentially modified by its exclusive owner, and a block in S state indicates the block is present in multiple private caches. The nature of M and S states is sufficient for SelectDirectory to identify whether the block is temporarily private or not. Specifically, a block in M state only has its tag entry allocated, while a block in S state have entries in both tag and data arrays. Therefore, SelectDirectory does not require any new coherence states. On the other hand, the downgrade action will minimally change the coherence protocol. When downgrading a block from the data array to the tag array due to a data eviction, we need to change the state of the block from S to M. Overall, SelectDirectory does not introduce complexity to the coherence protocol.

D. Latency

This section describes a comparison of latencies between conventional sparse directory and SelectDirectory. CACTI 6.5 [19] is employed to model the access latency using 32nm technology. For the conventional Sparse $2\times$, both parallel and serial tag and data lookup are considered, while we use serial lookup for SelectDirectory. Table I shows the latencies of Sparse $2\times$, $SD(2\times, 1/2\times)$ and $SD(2\times, 1/4\times)$. We assume both SelectDirectory configurations use a fully associative data array which gives us an upper bound on directory access latency. Thanks to the directory size reduction, the latencies of $SD(2\times, 1/2\times)$ and $SD(2\times, 1/4\times)$ are 12% and 22% lower than Sparse $2\times$ with serial lookup. Note that Sparse $2\times$ with parallel lookup can reduce the latency down to 0.51ns, but it still requires 2-cycle at 2GHz clock, which is the same as serial lookup. Given the energy increase due to parallel lookup, we will use serial lookup for the baseline Sparse $2\times$. Thus, we consider that the latency of SelectDirectory does not increase with respect to the conventional sparse directory. The same latency (2-cycle) is used for the baseline Sparse $2\times$ and all SelectDirectory configurations in our experiments.

TABLE I: Directory access latencies.

Directory architecture	Access latency (ns)
Sparse $2\times$, Parallel lookup	0.51
Sparse $2\times$, Serial lookup	0.82
$SD(2\times, 1/2\times)$	0.73
$SD(2\times, 1/4\times)$	0.64

E. Comparison with HR

L. Fang et al. recently propose Hybrid Representation (HR) [10] as a mechanism for exploiting the large fraction of temporarily private blocks. We identify HR to be the closest work to our proposal. For each directory set of HR, only a few entries have full sharer vectors (*vector entries*, VE) while the rest are only capable of tracking one sharer (*pointer entries*, PE). When a PE needs to track more than one sharer, the entry is moved to a VE if there is an unused VE. Otherwise a VE has to be converted into a PE to allow a swap between the PE and the VE. If the sharer number of the victim VE is less than a threshold, the sharer vector is rounded down to one current sharer. If not, a broadcast bit is set indicating that the block is potentially present in all private caches.

Compared with SelectDirectory, HR has several shortcomings: (**i**) as the locations of PEs and VEs are fixed, HR involves swapping between directory entries. It increases energy by swapping both the tags and data. In contrast, instead of doing swaps between entries, when a single-sharer entry (tag entry) becomes actively shared, the decoupled structure of SelectDirectory allows it to link to multi-sharer entries (data entries) using the forward pointer. (**ii**) Moreover, HR needs to ensure atomicity during a swap. Otherwise the vacant entries could be occupied by a request in interim, causing unexpected new races. (**iii**) When overflow happens in the VEs, imprecise representation is used to reduce the conversion overhead. However, this imprecision in coherence tracking can result in unnecessary private cache snoops and network bandwidth. For every PE with the broadcast bit set (we denote it as PE-B), a broadcast may happen in the future. SelectDirectory dose not use any imprecise representation and thus imposing no additional private cache probes.

IV. Methodology

In this section, we provide the simulation infrastructure and workloads used for our evaluation.

A. Simulation Environment

For evaluation of our proposal, we use the *gem5* simulator [17] with Ruby full-system mode enabled. Garnet [15] is used to simulate a 2D mesh network-on-chip. Detailed parameters of the simulated system are listed in Table II.

TABLE II: Simulation parameters.

Cores	32 in-order cores, 2 GHz
L1 Cache	Split I & D, 32KB, 4-way, 64B block, LRU, 1-cycle access latency
L2 Cache	Private, 256KB, 8-way, 64B block, LRU, 3-cycle access latency
L3 Cache	Shared, 32 MB (32 slices of 1 MB each), 16-way, 64B block, LRU, 20-cycle access latency
Baseline Directory	MESI coherence, sparse directory, explicit eviction notification, 2× provisioning ratio (32 slices of 8K entries each), 8-way, LRU, 2-cycle access latency
Network	2D Mesh, 16B-flit, 1/5-flit control/data packets, 5-stage router, 1-cycle link
Memory	2GB, DDR3, 16 channels

B. Workloads

As shown in Table III, we use PARSEC [4] and SPLASH-2 [18] workloads to evaluate our proposal. For stable and faithful measurements, we run each experiment multiple times and bind each thread to a particular core by invoking the Linux system function *pthread_setaffinity_np* where the threads are spawned. All workloads run correctly to completion.

TABLE III: Workloads and input size.

PARSEC	blackscholes, bodytrack, canneal dedup, fluidanimate, freqmine, swaptions, x264	simmedium
SPLASH-2	barnes fmm radiosity water_spatial cholesky fft lu radix	32K particles 64K particles BF refinement=1.5e-2 20^3 Molecules 13992x13992, NZ=316740 4M points 1Kx1K matrix, block=16 16M keys, radix=4K

V. Evaluation

We start the evaluation by exploring the SelectDirectory configurations to see how small it can be without sacrificing cache performance. Along with it, we also demonstrate the energy and area costs of the design. Section V-C demonstrates the worst case analysis for energy consumption. In Section V-D, we compare SelectDirectory with HR [10]. Then we proceed to extend SelectDirectory by incorporating the idea of state-of-the-art coherence directory [9].

A. SelectDirectory Configuration Exploration

By selectively allocating coherence data, the directory size can be significantly reduced, but excessive directory size reduction will cause significant back invalidations, which consequently hurt private cache performance. We explore to find a reasonable SelectDirectory configuration by measuring private L2 cache miss rate. Figure 3 shows the L2 cache miss rates of SelectDirectory with varying data array size and associativity. As the results illustrate, the cache performance impact is barely noticeable (less than 0.5%) when shrinking SelectDirectory from $SD(2\times, 1\times)$ to $SD(2\times, 1/4\times)$ across all workloads. However, downsizing the data array from $1/4\times$ to $1/8\times$ can experience a cache performance degradation of up to 13.7%. Meanwhile, increasing the associativity of the data array from 32-way to full-associativity only improves cache performance by 0.3% across all SelectDirectory configurations. Given the storage overhead of big forward pointers, the results render 32-way a desirable design point to reuse the owner pointer storage (5-bit) for the forward pointer. Unless otherwise stated, we will use 32-way for the data array associativity.

Figure 5 shows the rate change of back invalidation over the baseline for SelectDirectory with different provisioning sizes. Associating it with Figure 3 offers an insight on how back invalidations impact cache performance when reducing the size of SelectDirectory. In some cases (e.g. *blackscholes*), the back invalidation rate is insensitive to SelectDirectory size. Because of the low data sharing degree, these workloads can have an extremely small SelectDirectory ($SD(2\times, 1/16\times)$) with negligible performance penalty. On the contrary, a number of workloads see a rise in invalidation rate when using a $1/8\times$ data array. In the case of *barnes*, the invalidation rate increases by 14.9%, which explains its 13.7% cache performance drop in Figure 3.

The optimal SelectDirectory size for a single workload reflects its working *actively shared* data set size. The above results reveal that $SD(2\times, 1/4\times)$ can offer comparable cache performance to the conventional sparse directory for all workloads. We show the normalized execution time of $SD(2\times, 1/4\times)$ in Figure 4a. It behaves nearly identical to the baseline Sparse $2\times$. Note that this experimental result closely matches the analysis presented in Section II.

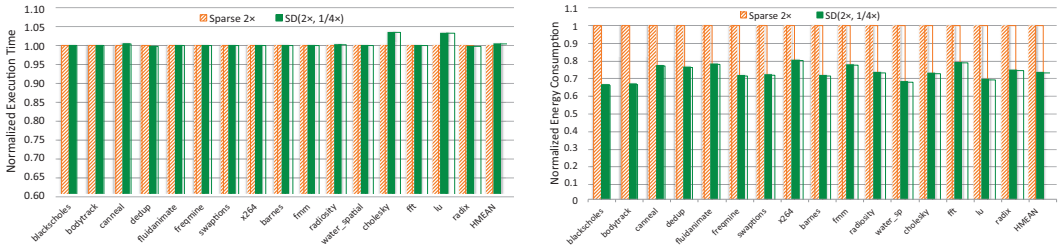
B. Energy & Area

CACTI 6.5 [19] is used for energy and area assessments assuming 32nm technology. The per-access energy of the directories is presented in Table IV. We also develop a model that measures the dynamic energy consumed in the NoC, L2 and L3 caches. For cache structures, the energy cost of coherence induced cache accesses is also accounted for. The NoC energy is obtained from DSENT [5].

Figure 4b shows the directory energy of $SD(2\times, 1/4\times)$ relative to Sparse $2\times$. On an average, $SD(2\times, 1/4\times)$ reduces the directory energy by 26.9%.

Back invalidations can hurt private cache performance, increase the network bandwidth and force more L3 accesses, thus the directory size reduction may affect the energy of L2, NoC and L3. Figure 6 shows energy consumed in NoC, L2

Fig. 3: L2 Miss rates of various SelectDirectory configurations, all normalized to Sparse 2×.



(a) Performance (b) Directory energy

Fig. 4: Performance and directory energy of $SD(2\times, 1/4\times)$, normalized to Sparse 2×.

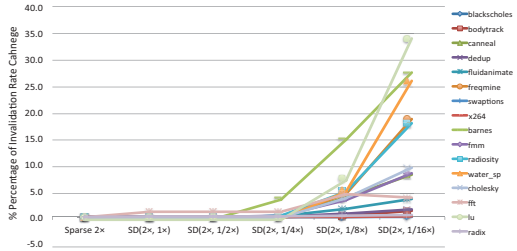


Fig. 5: Percentage change of back invalidation rate over the baseline for SelectDirectory with different provisioning sizes.

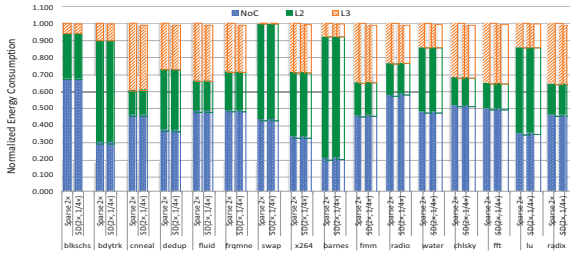


Fig. 6: Energy impact of $SD(2\times, 1/4\times)$ on NoC, L2 and L3 (w/o directory), normalized to Sparse 2×.

and L3 for $SD(2\times, 1/4\times)$, normalized to the baseline. As the figure shows, the size reduction of SelectDirectory does not impact the energy of these structures at all.

Area results are listed in Table V. As shown in the table, the data array of Sparse 2× occupies more than 2x area than the tag array. $SD(2\times, 1/4\times)$ reduces the area of the data array to 3.1x smaller than the tag array. As the tag array of SelectDirectory is extended by the forward pointer, it is slightly bigger than the baseline. Nevertheless, compared to Sparse 2×, SelectDirectory saves 2.04x total directory area.

TABLE IV: Energy overheads of $SD(2\times, 1/4\times)$.

Directory architecture	Energy per-access (pJ)		
	Tag array	Data array	Total (tag+data)
Sparse 2×	7.13	9.78	16.91
$SD(2\times, 1/4\times)$	8.06	2.24	10.30

TABLE V: Area overheads of $SD(2\times, 1/4\times)$.

Directory architecture	Area per-slice (mm^2)		
	Tag array	Data array	Total area
Sparse 2×	0.051	0.107	0.170
$SD(2\times, 1/4\times)$	0.058	0.019	0.084

C. Worst Case Analysis for Energy Consumption

For a single block, transitions between temporarily private and actively shared states will require upgrade and downgrade actions that consumes energy. We first calculate the energy overheads of the two actions for $SD(2\times, 1/4\times)$. As shown in Table VI, both of upgrade and downgrade consists of two tag array access and a data array access. For the former, the extra tag access is for setting the forward pointer, while for the latter the extra tag access is used for overwriting the forward pointer with the owner. Thus, for every upgrade or downgrade, $SD(2\times, 1/4\times)$ consumes additional 1.45pJ energy than the baseline. In the worst case scenario, every request will cause an upgrade or downgrade, exhibiting extreme migratory sharing patterns [16]. This worst case will consume 1.45pJ/16.91pJ=8.6% more energy compared to the baseline. However, in our experiments, the energy overheads in upgrade and downgrade actions are completely compensated by the energy reduction from SelectDirectory (26.9% energy reduction). Therefore, although the worst case has an 8.6% energy increase, SelectDirectory is highly resistive to it, benefiting from the energy-efficient design.

TABLE VI: Energy overheads of upgrade and downgrade actions for $SD(2\times, 1/4\times)$.

	Tag access	Data access	Energy (pJ)
Sparse 2×	1	1	16.91
Upgrade	2	1	18.36
Downgrade	2	1	18.36

D. Comparison with HR

We denote $HR(t\times, v/assoc)$ as the architecture for comparison, where t is the provision ratio of the tags and v is the number of VEs in a set ($assoc$ ways). According to [10], using a single VE in a set will cause enormous back invalidations, and a 2-VE configuration is used for better trade-offs. Thus, we compare $SD(2\times,\ 1/4\times)$ with $HR(2\times,$ 2/16), as the HR configuration asymptotically reduces the data array size by 8x compared with Sparse $2\times$, which is similar to $SD(2\times,\ 1/4\times)$. Figure 7 shows L2 cache miss rates of $SD(2\times,\ 1/4\times)$ and $HR(2\times,$ 2/16), all normalized to Sparse $2\times$. Results of $HR(2\times,$ 1/8) is also shown in the figure for reference. We can observe that having a single VE in a set is not sufficient for some workloads. $HR(2\times,$ 2/16) mitigates the problem, but the worst case still has 8.7% cache performance degradation. On average, $SD(2\times,\ 1/4\times)$ outperforms $HR(2\times,$ 2/16) by 1.2%.

Additionally, the swaps and high-associative lookups worsen the directory energy. As shown in Figure 8, $HR(2\times,$ 2/16) generates more energy than the baseline by up to 23.2%. In contrast, $SD(2\times,\ 1/4\times)$ is consistently more energy-efficient than the baseline across all workloads. On average, $SD(2\times,\ 1/4\times)$ consumes 28.3% lower directory energy than $HR(2\times,$ 2/16).

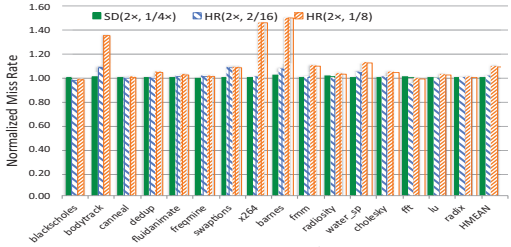


Fig. 7: L2 miss rates of $SD(2\times,\ 1/4\times)$, $HR(2\times,$ 2/16) and $HR(2\times,$ 1/8), normalized to Sparse $2\times$.

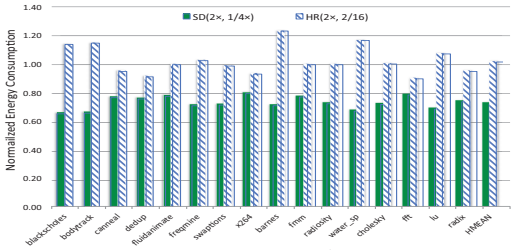


Fig. 8: Directory energy of $SD(2\times,\ 1/4\times)$ and $HR(2\times,$ 2/16), normalized to Sparse $2\times$.

E. Extending SelectDirectory with State-of-the-Art Coherence Directory

State-of-the-art directories use multi-grain directory entries (MGD) to exploit the temporarily private nature of large continuous chunks of blocks (or region) [2][9]. Since a region entry is temporarily private by nature, the design can naturally fit into our selective data allocation, further shrinking the tag array of SelectDirectory. We combine SelectDirectory with MGD (denoted as SD+MGD). Table VII shows the relative execution time and directory energy reduction of SD+$MGD(1/2\times,\ 1/4\times)$ to the baseline across all workloads. By leveraging multi-grain directories, the extended SelectDirectory further reduces the directory size and energy, with no statistically significant performance loss.

TABLE VII: Execution time and directory energy reduction of SD+$MGD(1/2\times,\ 1/4\times)$, normalized to Sparse $2\times$.

Directory architecture	Execution time	Directory energy reduction
$SD(2\times,\ 1/4\times)$	1.005	26.9%
SD+$MGD(1/2\times,\ 1/4\times)$	1.009	40.1%

VI. Conclusion

Applications have significant portion of temporarily private blocks in the cache hierarchy. We exploit this phenomenon by proposing a practical and effective directory design that decouples the tag array and the data array, and allocates data entries only for actively shared blocks. Experimental results reveal that our proposal can substantially save the directory storage, area and energy without sacrificing performance.

VII. Acknowledgments

This work was supported by Huawei International Pte. Ltd. research grant and Singapore Ministry of Education Academic Research Fund Tier 1 T1-251RES1120.

References

[1] "OpenSPARC T2 system-on-chip (soc) microarchitecture specification, may 2008."
[2] M. Alisafaee, "Spatiotemporal coherence tracking," in *MICRO'12*.
[3] A. Gupta et al, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *ICPP'90*.
[4] C. Bienia et al, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT'08*.
[5] C. Sun et al, "DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *NOCS'12*.
[6] D. Sorin et al, *A Primer on Memory Consistency and Cache Coherence*, 2011.
[7] H. Zhao et al, "SPACE: Sharing pattern-based directory coherence for multicore scalability," in *PACT'10*.
[8] J. Albericio et al, "The reuse cache: Downsizing the shared last-level cache," in *MICRO'13*.
[9] J. Zebchuk et al, "Multi-grain coherence directories," in *MICRO'13*.
[10] L. Fang et al, "Building expressive, area-efficient coherence directories," in *PACT'13*.
[11] L. Zhang et al, "SpongeDirectory: Flexible sparse directories utilizing multi-level memristors," in *PACT'14*.
[12] M. Ferdman et al, "Cuckoo directory: A scalable directory for many-core systems," in *HPCA'11*.
[13] M. Martin et al, "Why on-chip cache coherence is here to stay," *Commun. ACM, 2012*.
[14] M. Qureshi et al, "The V-Way cache: Demand based associativity via global replacement," in *ISCA'05*.
[15] N. Agarwal et al, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS'09*.
[16] N. Barrow-Williams et al, "A communication characterisation of splash-2 and parsec," in *IISWC'09*.
[17] N. Binkert et al, "The gem5 simulator," *Comput. Archit. News, 2011*.
[18] S. Woo et al, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA'95*.
[19] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A Tool to Understand Large Caches," University of Utah and HP Laboratories.
[20] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *HPCA'12*.