SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors

Abstract

Cache-coherent shared memory is highly desirable for programmability in many-core systems. Many directory-based schemes have been proposed, but highly-dynamic and non-uniform sharing make efficient directory storage challenging, with each giving up storage space, performance or energy.

We introduce SpongeDirectory, a sparse directory structure that exploits multi-level memristory technology. SpongeDirectory expands directory storage in-place when needed by increasing the number of bits stored on a single memristor device, trading latency and energy for storage.

We explore several SpongeDirectory configurations, finding that provisioning rate of 0.5x with memristors optimized for low energy consumption is the most competitive. This optimal SpongeDirectory configuration has comparable performance (within 1%) to conventional sparse directory, requires 20x less storage space, and consumes half the energy. In addition, SpongeDirectory is orthogonal to, and may be combined with, other proposals, such as compression techniques and SCD[29], for further savings.

1. Introduction

As the number of processor cores increases, multi-/many-core chips tend to favor directory- over bus-based snooping coherence for network-on-chip bandwidth considerations. Simple directory schemes based on sharer vectors are also not scalable, however, because storage increases super-linearly with the number of cores. Therefore, a storage efficient directory scheme is needed for future extreme-scale many-core system. Many directory schemes (e.g., [14][41][43][20][29]) have been proposed to reduce the storage requirements, at the cost of energy, latency or complexity. For example, duplicate-tag [14] and tagless [41] directories provide scalable performance and area, but they are not energy efficient. The recently proposed SCD [29] scheme is scalable, but only if it is coupled with an underlying, complex ZCache architecture[28].

We empirically show that the requirements for directory coherence storage are quite different among different applications—some applications (e.g., `radix`) have a large number of directory entries, but most directory entries have very few (e.g., one) sharers; some other applications (e.g., `raytrace`) have much fewer directory entries, but on average much more sharers in each directory entry. Using conventional SRAMs, it is quite complex to provide the flexibility for both situations while keeping overall storage overhead low [10][28].

In this paper, we seek to use emerging memristor technologies to solve this problem. Memristors offer promising characteristics for storage devices—high-density, non-volatile, low-energy electrical switching, CMOS-compatibility [35], and most importantly for this work, the ability to dynamically trade off read and write latency for storage density [3]. This variable-precision storage, projected to be up to seven bits[3] in a single memristor bit, is a perfect match to meet the requirement of different directory storage requirements in different situations—For situations where directory entries are many but each with few sharers, we use additional memristor levels to store extra directory entries; for situations where directory entries are few but each with many sharers, we use additional memristor levels to store extra sharer information of existing directory entries. Our optimal SpongeDirectory configuration shows that the extra latency has a negligible effect on overall system performance, it saves 20x in storage, and it consumes half the energy of a conventional directory.

The rest of the paper is organized as follows. We first give a background on memristors and cache coherence directory storage schemes in Section 2. Section 3 presents our Sponge Directory scheme. We then present our methodology and results in Sections 4 and 5. We present a more thorough related work section in Section 6, followed by our conclusions and future work in Section 7.

2. Background

The SpongeDirectory is built on two bodies of work – future multi-level memristor technologies and on-chip cache coherence directories. Here, we present the relevant information in those areas.

2.1. Multi-Level Memristor Storage-Latency Trade-offs

Memristors [38][39] are in the family of resistive memories. Others include Phase Change Memories (PCM) and Spin-Transfer Torque RAM (STTRAM). Memristors, when used as a direct replacement for SRAM, are projected to be significantly more dense than conventional SRAM. Because memristors are analog devices, they can be used for analog computations or, depending on the precision of the storage, storing multiple digital bits in one cell [3]. We propose to exploit this latter characteristic (trading dynamic density for energy savings).

A memristor cell [38] is a device whose resistance can be changed in a wide range. Holding a voltage above a specific threshold for enough time (i.e., a write pulse) changes the resistance of the memristor cell. An Analog-to-Digital Circuit (ADC) reads the resistance as a digital value. Depending on the ADC setting, the same resistance of a memristor cell could be interpreted as a different number of bits [13][37]. Figure 1 shows that the same resistance can be interpreted as

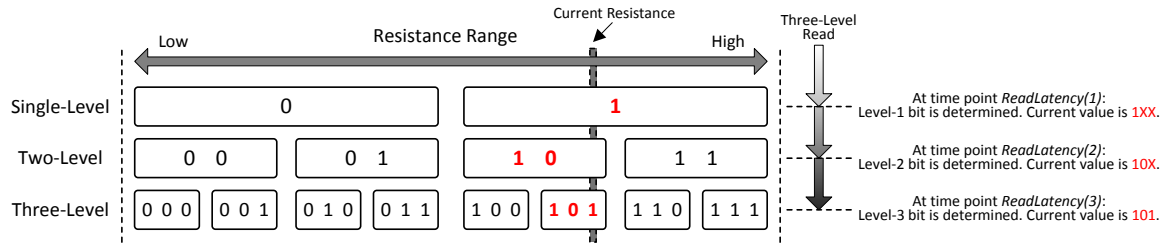

Figure 1: Multi-level memristor cell, and the step-wise manner of a multi-level memristor read. The same resistance can be read as different numbers of bits, and the most significant bits are read first.

b'1 when the ADC is using one-level accuracy, and as b'10 and b'101, respectively, when the ADC is using two-level and three-level accuracy. Alibart et al. [3] recently showed that a single memristor cell can be used to store up to seven bits of data.

The higher precision of the memristor operations come at a cost in latency and energy. Here we describe in detail the characteristics of multi-level memristor operations:

2.1.1. Multi-level Memristor Reads As the number of bits being stored in a single memristor increases, the ADC must distinguish among more values, and the range of resistive values mapping to a single digital value decreases. Therefore, more precise read operations require longer latency.

Another interesting and important characteristic of multi-level memristor read operation is its asymmetry to different bits. Because of this, we will refer to the different bits stored in a single memristor as being stored in different *levels*. Because the most significant bit requires much less precision to be read out than the least significant bit, we refer to the most significant bit as being the shallowest bit, or in the shallowest level, and the least significant bit is the deepest bit, or in the deepest level.

In this paper, we assume a successive approximation ADC [6] which allows us to read out multi-level memristor data in a step-wise manner. If ReadLatency(n) is the time required to read the nth most significant bit, a three-level read operation will be complete at *ReadLatency(3)*. As Figure 1 shows, however, the data in the first level (i.e., the most significant bit or shallowest level) can be read out by *ReadLatency(1)*, and the data in the second level can be used by *ReadLatency(2)*.

The important result is that if we only need to read out the most significant bit of a multi-level memristor, we pay the latency and energy of a less precise memristor.

2.1.2. Multi-level Memristor Writes In order to write very precise resistance values, multi-level memristors use an iterative write method [3]. That is, a multi-level write is composed of multiple iterations, each iteration consisting of a multi-level read followed by a write pulse.

As we will see in Section 4, a multi-level write can have a very considerable latency (up to hundreds or even thousands of nanoseconds). Luckily, though, the operation can be interrupted between iterations to service more critical operations (e.g. a read).

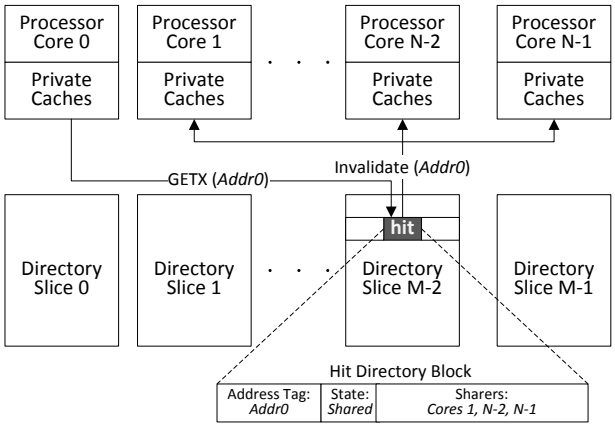

Figure 2: An example of a sparse directory scheme used in a multi-/many-core processor to track private cache coherence information. A sparse directory uses set-associative cache-like structures and is is often distributed into slices (M slices in this figure). For a GETX request for exclusive access on memory block $Addr0$, address hashing determines the appropriate slice (M2), all other sharer cores' private caches are invalidated, and requester gets exclusive access.

In conclusion, multi-level memristor operations trade operation latency and energy for storage density. Using multiple levels incurs high latencies and energies, but we are able to mitigate these effects with careful data organization and scheduling. The asymmetry in read latency for different levels allows us to access some data at low latency and energy. Finally, while the write latency appears unacceptably high, we can suspend the write while performing more time-critical operations.

2.2. On-chip Coherence Directories

There are several baseline directory architectures used for on-chip coherence, such as sparse directory, duplicate tag directory, and in-cache directory. A duplicate tag directory has been shown not to be energy scalable (since its associativity is linear with number of cores). An in-cache directory requires a special inclusive shared *Last Level Cache* hierarchy. Therefore, among the possible baseline directory architectures, the sparse directory [16] is general-purpose and requires the least energy. Recently, a number of proposals (e.g., [12][29][8][4][11]) work on improving the scalability

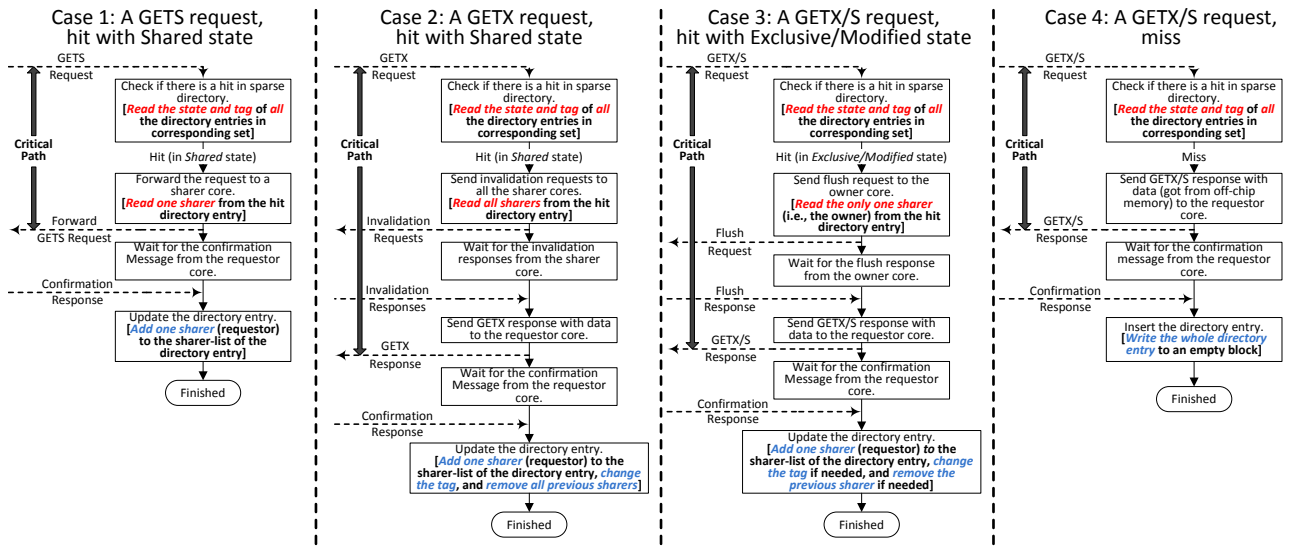

Figure 3: Behaviors of GETX and GETS requests in a conventional sparse directory using MESI protocol showing reads (red) and writes (blue). Goal: reduce latency for common operations on critical path. We see that: (1) Only reads are on the critical path. (2) All reads require state and tag, and most require one sharer. (3) Very few operations require reading multiple sharers.

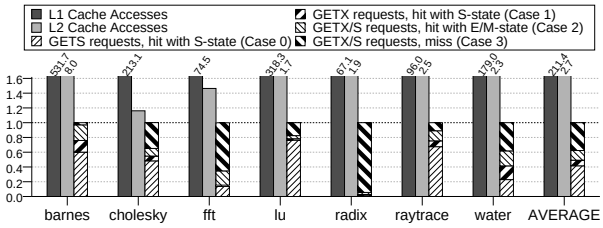

Figure 4: Number and type of memory accesses normalized to total directory accesses showing that directory accesses are much less frequent than cache accesses, allowing latency tolerance of such operations.

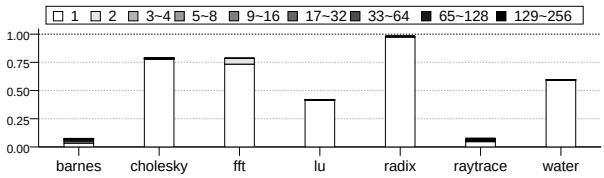

Figure 5: Number of directory entries with different sharers, normalized to the number of cache blocks showing variance in behavior between many entries (radix) and many sharers (raytrace).

of sparse directories, each with its own trade-off among complexity, storage and performance. Our scalable design provides a new design point in these trade-offs. Specifically, with the help of emerging multi-level memristors, our technique has the lowest storage requirement among all the proposals with a gentle degradation in performance for rare cases.

Figure 2 provides a logical view of on-chip sparse directory coherence, consisting of a logical uniform directory module and multiple processor cores along with their private caches. To avoid directory and network hot-spotting, the directory module is often divided into multiple directory slices. Each directory slice contains multiple directory entries, typically organized in a set-associative manner, just like caches. Each directory entry usually contains at least the following fields: address tag (recording the address of correspondent memory block), state (three states in a conventional MESI protocol) and sharers (recording which cores currently have the correspondent memory block).

If a coherence processor core asks for a coherence memory block which does not exist on its private caches, it will issue a coherence request (e.g. GETX in Figure 2) to the sparse directory. The sparse directory then performs corresponding actions (e.g., invalidating all the cached copies of the memory block). The downside of the original sparse directory is its storage requirement, caused by the requirement that it hold all information perfectly. Depending on who is sharing what, there are two sources of storage inefficiency:

- **Storing sharer information in an entry:** If in a sharer-vector form, the storage of such information is linear with the number of cores and thus not scalable.
- **Providing sufficient associativity:** In theory the number of valid directory entries should never exceed the number of private cache entries. However, due to the set-level and slice-level non-uniformity of sparse directory, the sparse directory often needs to over-provision (usually 2× of the number of private cache entries [7]) the directory entries. The way to solve such a problem usually lies in increasing the actual associativity of the sparse directory (e.g., Cuckoo directory [12]).

Here we make an analysis of a conventional 256-core sparse directory (i.e., the *ConvDir* described in Section 4), to shed light on designing low-cost and efficient many-core coher-

ence directory using multi-level memristors. This analysis is conducted from both spacial and temporal directions.

2.2.1. Directory Design: Timing In order to exploit the fact that multi-level memristor reads are asymmetric, we need to analyze the timing of traditional conventional sparse directories to determine which items are both commonly accessed and on the critical path, and thus should be in the shallowest levels of the memristors. Figure 3 presents detailed flow graphs of GETX/GETS operations in a sparse directory. Figure 4 shows the number of directory requests with respect to L1 cache and L2 cache requests. Using this information, we make the following set of observations that will guide our design:

- For a typical MESI-based directory coherence system, there are four types directory requests: Exclusive Read (i.e., GETX), Shared Read (i.e.,GETS), Exclusive Write-back (i.e.,PUTX) and Shared Write-back (i.e., PUTS). While GETX and GETS are on the critical path of the execution, PUTX and PUTS are not. *Therefore, we should optimize the latency of GETX and GETS operations.*
- From Figure 3, the read operations (in red) are on the critical path of the directory requests, but the write operations (in blue) are not on the critical path. *Therefore, low read latency is more important than write latency.*
- From Figure 3, for all reads, the sparse directory needs to read the tag and state information of all entries in corresponding directory set. On a hit, it usually (cases 1 and 3) needs to read only one sharer from the hit directory's entry. From Figure 4, Case 2 happens relatively rarely. *Therefore, low read latency of tags, states and one sharer is more important than read latency of additional sharers*
- From Figure 4, the number of directory requests are on average over 210× and 1.7× less than the number of L1 cache requests and L2 cache requests, respectively. According to Amdahl's law, *a moderate degradation of directory performance is unlikely to affect overall system performance.*

2.2.2. Directory Design: Organization For each evaluated benchmark, we take a snapshot when the total number of sharers in the directory entries is the largest. Figure 5 is a histogram of how many directory entries have each number of sharers. We can see that benchmarks vary widely in their maximum sharing requirements.

- Some benchmarks (e.g., radix) have a large number of directory entries, but almost all the directory entries are owned by only one sharer. *These benchmarks require more directory entries, but seldom require more sharer storage for each directory entry.*
- Some other benchmarks (e.g., raytrace) have much fewer directory entries, but some of the directory entries have many sharers. *These benchmarks require fewer directory entries, but some of the directories require more storage for sharers.*

Therefore, our proposed directory scheme should be flexible enough to handle both situations - when many directory entries are required and when many sharers are required for some directory entries.

3. SpongeDirectory

In this section, based on the analysis of both multi-level memristors and sparse directories in Section 2, we propose the detailed architecture of SpongeDirectory. Without the loss of generality, our proposed architecture is for a 256-core system (as described in Section 4).

The storage of each SpongeDirectory slice, just like a normal sparse directory slice, is a N-way set-associative memory (N refers to the associativity). Therefore, each set of SpongeDirectory has N *blocks* in it. The difference is that the memory is made of multi-level memristor RAMs, therefore each *block* is able to be configured with flexible number of levels. We call each level of a SpongeDirectory *block* a SpongeDirectory *item*.

3.1. Identifying Level Count of a SpongeDirectory Block

The SpongeDirectory must be able to identify the number of *items* in each *block*. Intuitively, we need to store item count information of all SpongeDirectory *blocks* into a small peripheral RAM. However, this will complicate the design of SpongeDirectory. Here we propose a low-cost but effective mechanism, a *usage bit*, that encodes the number of items stored in a single memristor bit whose value is read only to the deepest filled item.

As is shown in Figure 6, each SpongeDirectory *block* has one *usage bit*. Figure 7 shows a working example of a *usage bit*—if the correspondent *item* is the deepest valid *item* of the block (e.g., the Level 4 *item* in Figure 7), its *usage bit* is assigned as 0; otherwise, it is assigned as 1. Therefore, A multi-level memristor read can stop once it reads a *usage bit* with value 0.

3.2. SpongeDirectory Entry Formats

A *directory entry* refers to the directory information of a cacheline block. There are two types of *items* - *head items* and *body items*. A *head item* is either invalid or contains the tag, state, and at least one sharer. A new *directory entry* needs only one *head item*, since it only needs its tag, address, and the one sharer that brought it in. As additional cores share the cacheline, additional *body items* are added to store the additional sharers.

We use two different *directory entry* formats for storing sharer information. First, the *sharer pointer* scheme stores information only for sharers, allowing the number of items to grow as the number of sharers grows, but requiring 8 bits per sharer. Second, the *sharer vector* scheme uses only 1 bit

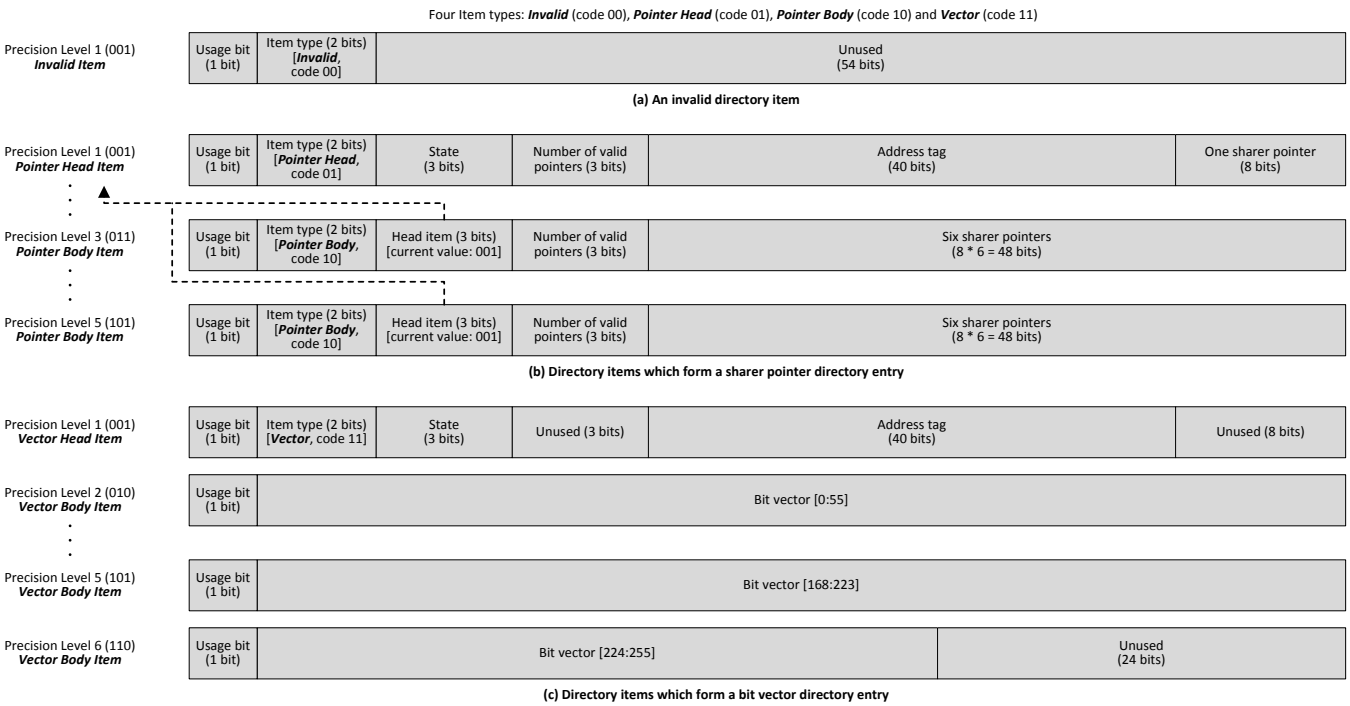

Figure 6: The entry formats of SpongeDirectory

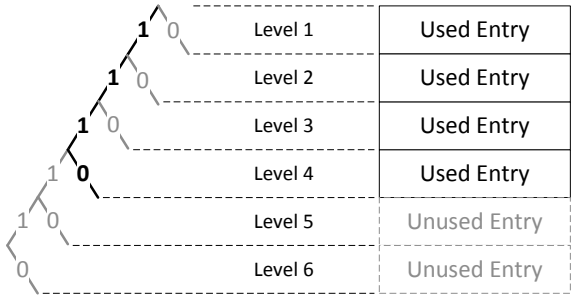

Figure 7: Storing the number of filled levels in a single multi-level bit

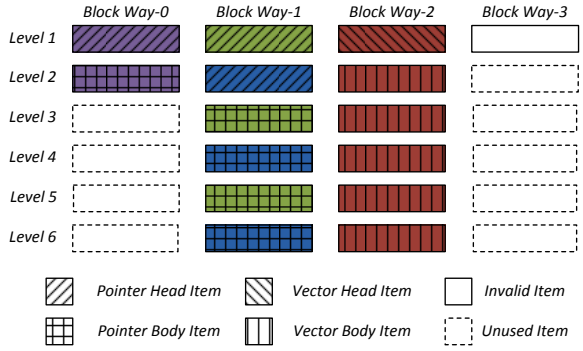

Figure 8: A snapshot of a 4-way SpongeDirectory set.

per sharer, but requires space for all cores (not just those sharing it). This is efficient for many sharers. Similar to prior work[29], we use a hybrid technique to obtain the advantage of both *sharer pointer* and *sharer vector* schemes.

In order to simplify the design:

- One *block* can hold up to six *items* (out of a possible seven [3]).
- A single *block* may contain several *directory entries* in sharer pointer format.
- All *items* for the same *directory entry* must reside in the same block to simplify accesses.
- *Head items* are stored in the shallowest levels for fast completions of cases 1, 3, and 4 in Figure 3.
- A *directory entry* in *sharer vector* format uses the entire *block*.

Figure 6 shows the different formats and how they are utilized with different scenarios.

- *Invalid Items*. Figure 6(a) shows an *invalid item*. This is only used when the block contains no items.
- *Sharer Pointer Head and Body Items*. Figure 6(b) shows an example of a multi-item directory entry using the *sharer pointer* format. The *pointer head item* contains the address tag, state and one sharer pointer. The two *pointer body items* contain all other sharer pointers. Because multiple *directory entries* can be mapped to the same *block*, each body item has a pointer to its head item.
- *Sharer Vector Head and Body Items*: Figure 6(c) shows an example of a *directory entry* using the *sharer vector* format. A *sharer vector directory entry* has a fixed storage requirement for sharer information (in our case, 256 bits, requiring five *items*). With its *head item*, a *sharer vector directory entry* requires all six *items* of a *block*.

3.3. Minimizing Write Energy

As will be shown in Section 4.4, multi-level memristor writes require significantly more energy than other operations. In addition, it is often the case that just a few bits of an item, and therefore block, need to be changed. In order to save energy consumption, SpongeDirectory only writes the memristor bits of a block that need to change.

- **Inserting or removing *items*** If an entry is added or removed, because inserting or removing items cross the threshold of the number sharers in an item in *sharer pointer format*, then the entire *block* is written.
- **Adding a sharer within an *item* in *sharer pointer format*** requires only modification of the target *sharer pointer* and the *pointer count field.*.
- **Removing a sharer within an *item* in *sharer pointer format*** Because the pointer being removed may leave a hole, we may need to move a sharer pointer. Therefore, the maximum is two *sharer pointers* and the *pointer count field.*.
- **Adding or removing a sharer** in *sharer vector format* requires modification of only a single bit.
- **Changing directory entry format** requires the entire block to be written (see below).

3.4. Changing Directory Entry Format

Every *directory entry* is in *sharer pointer* format (with one *head item* and no *body items*) when inserted into the directory, and *sharer pointer items* are added and removed as needed. When the number of sharers surpasses a threshold, SpongeDirectory will upgrade a *sharer pointer directory entry* into *sharer vector* format. This requires us to remove all other *directory entries* that are sharing the same *block*, evicting those entries from the directory, translate the sharer pointers to vector locations, and write the *sharer vector items* of the upgraded *directory entry*.

Ideally, as the number of sharers falls below another threshold, a downgrade operation should occur to switch the *sharer vector directory entry* back into *sharer pointer* format. Such operations are expensive (latency and energy), however, and only beneficial when the change in sharers is large. All operations other than a GETX result in a change of only one sharer. When a GETX occurs, the resulting directory entry has only one sharer (the requester), so compressing the entry to a single item rather than the entire block is worthwhile. Therefore, we only downgrade a *sharer vector directory entry* when a GETX operation occurs. Our experiments show no performance loss as a result of restricting downgrades to this circumstance.

4. Methodology

4.1. Base Configuration

We implement our simulation platform with the GRAPHITE simulator [23], based on Pin [5]. As described in Table 1, we simulate a 256-tile cache coherent many-core system which is distributed in a 16×16 mesh network-on-chip. Each tile has split 32KB L1 I/D-caches, a private 256KB L2 cache, and a sparse directory slice.

To take into account the execution time variability of parallel benchmarks [2], we run each simulation multiple times and report the average and standard deviation of each collection of measurements. We evaluate our proposal using seven SPLASH-2 benchmarks [34]. As is indicated by Figures 4 & 5, these benchmarks exhibit a wide variety of directory behaviors.

4.2. Directory Architecture

As shown in Table 2, we model three sizes of SpongeDirectory architectures (*SpongeDirSmall*, *SpongeDirMid* and *SpongeDirLarge*), as well as a conventional sparse directory architecture using a sharer-vector scheme (*ConvDir*). We also calculate the storage requirement of the SCD architecture. We define the *provisioning rate* as the total number of *directory blocks* divided by the total number of cache blocks in all private caches. Each *ConventionalDirectory block* can hold only one *directory entry*, whereas each *SpongeDirectory block* can be configured to hold multiple *directory entries*. Therefore, *SpongeDirectory* needs a much smaller *provisioning rate* than conventional directories. In our experiments, we use a typical *provisioning rate* of $2\times$ for *ConvDir* and $0.25\times$, $0.5\times$ and $1\times$ respectively for *SpongeDirSmall*, *SpongeDirMid* (our default configuration) and *SpongeDirLarge*.

We can see that, thanks to the pointer-vector hybrid scheme, both SpongeDirectory and SCD[29] architectures manage to have much smaller block sizes than *ConvDir*; and, due to its tighter *item* organization, SpongeDirectory requires a smaller block size than SCD. Also, the default-sized SpongeDirectory (*SpongeDirMid*) requires only $0.5\times$ provisioning, compared with $1.0\times$ provisioning needed by SCD. As a consequence, the storage requirement of *SpongeDirMid* is over 20 times smaller than that of a conventional sparse directory and less than half that of a state-of-the-art SCD. Moreover, SpongeDirectory does not rely complex Zcache-like infrastructure.

4.3. Single-Level Memristor and SRAM-Cache Modeling

For the detailed modeling of memristors, we use nvsim [9] to obtain the read latency and energy of single-level memristor RAMs under at 22nm process. Cacti 6.5 is used to obtain energy characteristics of SRAM structures. Prior work [36] demonstrated that in designing the memristor-based RAM, there is a trade off between latency, energy and area. We explore two design points: *Latency-Optimized* and *Energy-Optimized*.

As Table 3 shows, memristors in the *Latency-Optimized* configuration are almost 2 times faster than in the *Energy-Optimized* configuration, but they consume almost 10x the

Table 1: Base configuration.

Frequency	1GHz
Processor	in-order, x86-64 ISA, IPC equals to 1 except on memory accesses, 64-byte cacheline size
L1 Caches	private split 32KB I/D-caches per processor, 4-way set-associative, 2-cycle latency, 0.21pJ tag lookup, 22.18pJ data access.
L2 Caches	private, 256KB per processor, 8-way set-associative, 11-cycle latency, 0.49pj tag access, 13.52pJ data access.
Coherence Protocol	MESI protocol, split request-response, with forward, sequential consistency.
Directories	Critical directory access latency = data path latency (5 cycles) + critical RAM operation latency + network/memory latency. ConvDirectories: read: 1 cycle (1ns), SpongeDirectories: See Sections 4.3 and 4.4
Network	16 × 16 MESH network, 4 cycles per hop. 1 process/hierarchy per tile.
Memories	350 cycle latency.

Table 2: Compared directory architectures. SpongeDirectory formats in Figure 6.

	Directory type	RAM type	Directory block organization	Provision rate	Directory RAM storage (data cache RAM storage is 360 KB per tile)
SpongeDirSmall	SpongeDirectory	memristor	4-way, 256 sets	0.25×	57 bits / block (6 levels/bit) -> 7KB (1.94% cache)
SpongeDirMid	SpongeDirectory	memristor	4-way, 512 sets	0.5×	57 bits / block (6 levels/bit) -> 14 KB (3.80% cache)
SpongeDirLarge	SpongeDirectory	memristor	4-way, 1024 sets	1.0×	57 bits / block (6 levels/bit) -> 38 KB (7.78% cache)
ConvDir	Conventional Sparse Directory	SRAM	4-way, 2048 sets 4 ways and 2048 sets	2.0×	299 bits / block: 3-bit state, 40-bit tag, 256-bit vector 299 KB per tile (= 83.1% of the cache storage).
SCD	Scalable Coherence Directory [29]	SRAM	Zcache-like [28], 4 ways and 1024 sets	1.0×	71 bits / block: 3-bit state, 40-bit tag 2-bit type, 26-bit sharer info field 35.5 KB per tile (= 9.86% of the cache storage).

Table 3: Characteristics of the modeled directory storage.

Architecture	Size	Storage type	1-level read latency		1-level read energy		Area	
			latency-optimized	energy-optimized	latency-optimized	energy-optimized	latency-optimized	energy-optimized
SpongeDirSmall	7KB	memristor RAM	1.621ns	4.424ns	36.8448pJ (with 57*4 bits)	2.6448pJ (with 57*4 bits)	9735.2um^2	2872.3um^2
SpongeDirMid	14KB	memristor RAM	1.633ns	4.746ns	36.9132pJ (with 57*4 bits)	2.7360pJ (with 57*4 bits)	9979.0um^2	3205.2um^2
SpongeDirLarge	28KB	memristor RAM	1.670ns	5.439ns	37.2324pJ (with 57*4 bits)	3.3972pJ (with 57*4 bits)	10404.7um^2	5671.0um^2
ConvDir	299KB	SRAM-based 4-way cache	« 1.000ns		tag: 1.3pJ (with 43 bits) data: 27.05pJ (with 256 bits)		466064um^2	

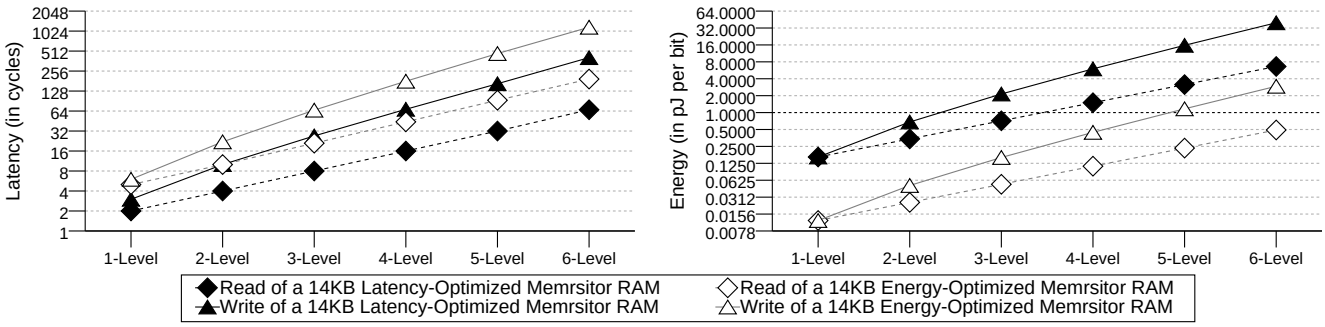

Figure 9: The latency and energy consumption of 14KB multi-level memristor RAMs used in *SpongeDirMid*.

energy and require more area. In addition, we see that for the *Energy-Optimized* configuration, all SpongeMemory configurations consume substantially less energy than a conventional directory when accessing the lowest level.

4.4. Multi-Level Memristor Modeling

Previous work on multi-level PCM reading technology [21] shows that the read latency increases *exponentially* with the number of bits stored in the PCM cell. For a multi-level memristor, we model the same read latency trend—Equation 1 shows how the latency is computed in a multi-level read. *ReadLatency(n)* refers to the latency of an n-level read. *ExpoBase* refers to the exponential base of the modeled multi-

$$ReadLatency(n) = \lceil ReadLatency(1) * ExpoBase^{n-1} \rceil \quad (1)$$

$$WriteLatency(n) = \lceil (ReadLatency(n) + WritePulseLat) * n \rceil \quad (2)$$

$$ReadEnergy(n) \approx ReadEnergy(1) * ExpoBase^{n-1} \quad (3)$$

$$WriteEnergy(n) \approx ReadEnergy(n) * n \quad (4)$$

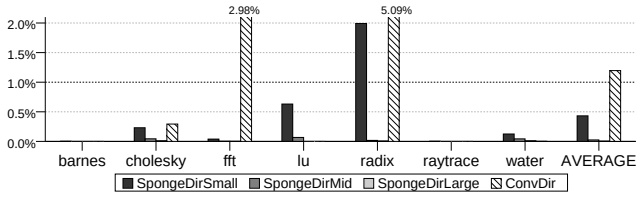

Figure 10: Directory Eviction Rate

level memristor technique.

As is mentioned in Section 2.1.2, a multi-level write is composed of multiple iterations, with each iteration consisting of a multi-level read followed by a write pulse. Alibart et al. [3] show that, for a multi-level write, its number of iterations grows linearly with the number of write levels. Therefore, we mode the same trend in Equation 2. *ReadLatency(n)* refers to the latency of an n-level read, and *WritePulseLatency* refers to the latency of a write pulse.

A write pulse consumes trivial energy compared to a read pulse [31], so we make an approximation that the energy consumption of a memristor operation equals the energy consumption of its read sub-operations. Since the power consumption of read operation is constant (because the read voltage is constant), we then have the Equations 3 & 4 for energy consumption of multi-level read/write operations.

For *ExpoBase*, we use the empirical number 2.1, which is derived from Alibart et al.'s work [3]. Work Memristor write pulses can be less than one nanosecond [31], therefore we assign $WritePulseLatency = 1ns$ in Equation 2. Figure 9 shows the latency and energy consumption of multi-level operations on 14KB memristor RAMs (with Latency-Optimized or Energy-Optimized configurations).

5. Results

In this section, we first present the performance of SpongeDirectory in terms of eviction rate, critical directory read operation latency, overall execution time, directory energy consumption, etc. These experiments utilize the different SpongeDirectory sizes (*SpongeDirSmall*, *SpongeDirMid*, *SpongeDirLarge*) with memristors optimized for low energy consumption. In addition, head items in a block are always stored in shallower levels than body items.

We then perform sensitivity analysis on the SpongeDirectory configurations, including (1) Energy-Optimized versus Latency-Optimized memristor designs, and (2) head-shallower versus first-come-first-serve item arrangement policies.

5.1. Eviction Rate

Eviction rate is the most straight-forward standard for evaluating the effectiveness of a directory scheme. Given a particular configuration and storage capacity of the cache, evictions occur when there is no place in the desired set to place a new entry.

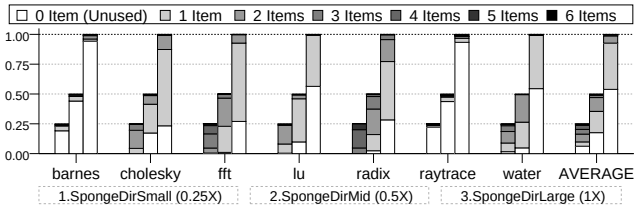

Figure 11: SpongeDirectory Blocks with Different Items, Normalized to Number of Cache Blocks

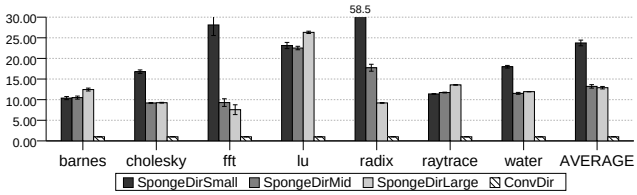

Figure 12: Average Latency of Critical RAM Read Operations (cycles)

We can see from Figure 10 that, even with a provisioning rate of only 0.25×–1×, SpongeDirectory manages to have negligible eviction rate (all less than 2%).

A interesting phenomenon is that, for some benchmarks (e.g., `fft` and `radix`), even a 0.25×-provisioned SpongeDirectory (*SpongeDirSmall*) achieves a much lower eviction rate than 2×-provisioned conventional directory. The reason is that the conventional directory experiences conflict misses in the sets, which SpongeDirectory handles by increasing the number of levels it uses (at the cost of latency and energy consumption). As we will see in Section 5.3, this advantage sometimes helps SpongeDirectory to achieve higher performance than conventional 2×-provisioned sparse directories.

Evictions occur in SpongeDirectory for two reasons. The first is due to upgrading directory entries from the sharer pointer format to the sharer vector format. In our initial design, this causes all other entries in the same block to be evicted (see Section 3.4). We could avoid such evictions by reinserting the entries in different blocks, but the small eviction rate may not merit this complication. The second is that the directory entries are sometimes non-uniformly distributed [12], leading to evictions in the hotspot directory sets. Such a phenomenon is relatively obvious at the lowest provisioning rate (i.e., *SpongeDirSmall*). However, even the eviction rate experienced by *SpongeDirSmall* is low, and we shall see that it has a negligible performance impact.

5.2. Critical Read Operation Latency

As discussed in Section 2.2, the performance of a directory schemes is very dependent on critical reads. In Figure 12, we present the latency of critical read operations of different directory architectures. Compared to a conventional sparse directory, SpongeDirectories require considerably longer latency (on average 22.75 cycles for *SpongeDirSmall*, 12.22 cy-

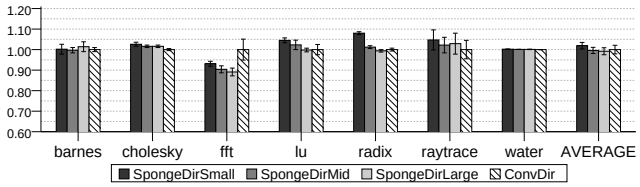

Figure 13: Normalized Execution Time

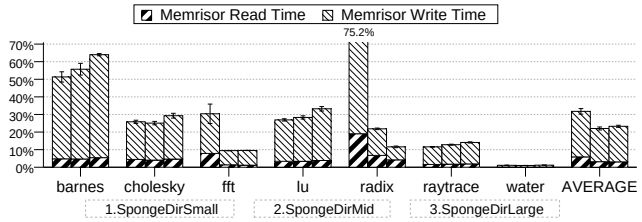

Figure 14: Percentage time busiest SpongeDirectory slice busy.

cles for *SpongeDirMid* and 11.90 cycles for *SpongeDirLarge*) to perform critical read operations. This longer latency happens because of two reasons:

- As an emerging technology, memristor devices still have much longer access latency compared with matured SRAM devices (see Table 3).
- Multi-level operations on memristor RAM further increases the average latency. Although we carefully designed the architecture of SpongeDirectory to reduce the frequency of multi-level memristor operations on the critical path, they cannot be fully eliminated. The lower provisioning rate implies SpongeDirectory sometimes needs to use multiple levels, especially for the benchmarks with many directory entries (e.g., `radix`).

Figure 11 illustrates the latter effect. As the provisioning rate decreases, the number of blocks used, and number of items within the blocks used, increases.

- Under 0.25× provisioning rate (*SpongeDirSmall*), many SpongeDirectory *blocks* utilize deep levels (e.g., for `radix` 61.32% of the *blocks* use 4 levels), increasing latency and energy consumption.
- Under 1× provisioning rate (*SpongeDirLarge*), many *blocks* are unused (on average, 53.87%). This is a waste of storage.

SpongeDirMid is a balanced choice, utilizing deep levels seldom but leaving few empty *blocks*. We will see that execution time is only minimally affected by the higher latencies.

5.3. Overall Execution Time

Figure 13 provides a comparison of overall performance between the SpongeDirectory and conventional sparse directory. This represents two competing effects - higher evictions in the conventional sparse directory versus higher read latencies in the SpongeDirectory.

Overall, SpongeDirectory is competitive despite its lower provisioning and more compact format (resulting in 20x fewer bits). Even *SpongeDirSmall* is, on average, only 1.7% slower (including an 8.0% increase for `radix`) than *ConvDir*. *SpongeDirMid* and *SpongeDirLarge* have similar performance as *ConvDir*.

Closer inspection of the eviction rates and levels used within SpongeDirectory explain the variance in the results. `fft` has a much lower eviction rate in the SpongeDirectory than the conventional directory, and higher levels are seldom used, so all SpongeDirectory configurations experience a performance gain. `radix` also has a lower eviction rate in the SpongeDirectory, but because most directory requests encounter misses in `radix` (Figure 3), the preserved directory entries do not improve performance.

Overall, this shows that the higher read latency is, in general, acceptable. The slowdown in *radix* shows that we should use deeper levels very sparingly, but they are sometimes very useful. Fundamentally, this tolerance for increased read latency is because, as discussed in Section 5.2, the read operation is one short step of a whole directory operation. Once a directory operation occurs, there is a long latency operation to follow—forwarding the request to another cache, visiting off-chip memory, invalidating the cached data, etc. This means that a small increase in the critical-path read step does not lead to a considerable difference in runtime (within reason).

5.4. Total Memristor Operation Time

With the help of suspendable multi-level memristor writes, long-latency non-critical memristor operations typically do not slow down critical read operations. However, this is based on a premise that there is enough time when the RAM is not performing critical reads to fit in non-critical, long-latency writes. If the memristor RAM is very busy, the critical requests might be blocked because the limited directory processing buffers are all taken by awaiting non-critical requests.

Fortunately, such a situation rarely happens. Figure 14 shows the memristor operation time in the busiest SpongeDirectory slice. We can see that, with 0.5× provisioning rate (*SpongeDirMid*), the memristor operation time takes only, on average, 22.0% of the overall execution time, with a maximum of 55.7% for `barnes`.

Our investigation into `barnes` revealed that one particular *sharer vector directory entry* is facing a large volume of directory requests. If desired, a future design could combine multiple operations in the write buffer into a single write to the *sharer vector directory entry*.

In addition, we will see in the sensitivity analysis that the execution time is resilient to small changes in read latency. Therefore, if the read must very occasionally wait for a write to end, it is unlikely to impact overall performance.

5.5. Dynamic Directory Energy Consumption

Figure 15 shows the breakdown of dynamic energy consumption of SpongeDirectory and a conventional cache.

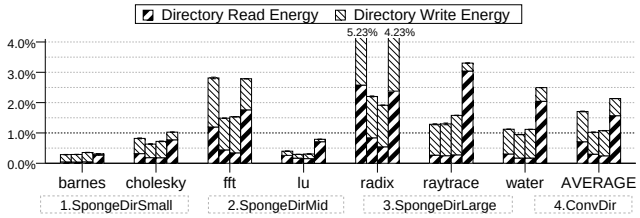

Figure 15: Percentage of on-chip memory dynamic energy

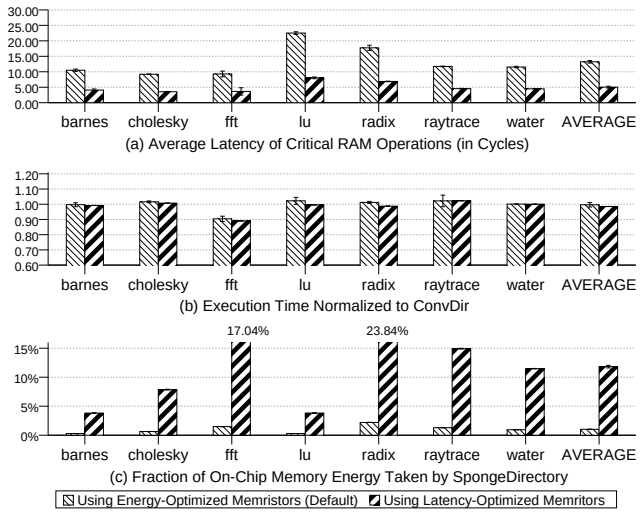

Figure 16: Latency-Optimized vs. Energy-Optimized Memristors

We see that, on average, all of the modeled SpongeDirectory configurations consume less energy than a conventional directory due to their smaller sizes. If we compare the three configurations, we see that *SpongeDirSmall* consumes substantially more power than *SpongeDirMid* and *SpongeDirLarge* for fft (92% more) and radix (145% more). This is due to increased accesses of deeper memristor levels. In addition, *SpongeDirLarge* often, but not always, consumes more energy than *SpongeDirMid*. This is because *SpongeDirMid* seldom uses deeper levels, and a *SpongeDirLarge* access of the same level consumes more energy. If we cross-reference this data with Figure 11, we see that the radix is the only application for which *SpongeDirLarge* consumes less energy than *SpongeDirMid* due to more items being stored in higher levels of the blocks.

5.6. Sensitivity to Memristor Design

As is discussed in Section 4.3, there are some trade-offs in designing the memristor RAMs. Our simulations so far used an *Energy-Optimized* design, optimized for low energy consumption. We now compare this to a *Latency-Optimized* design that is optimized for low latency operations. Figure 16 shows the read latency, performance, and energy consumption of the *Latency-Optimized* design normalized to the *Energy-Optimized* design.

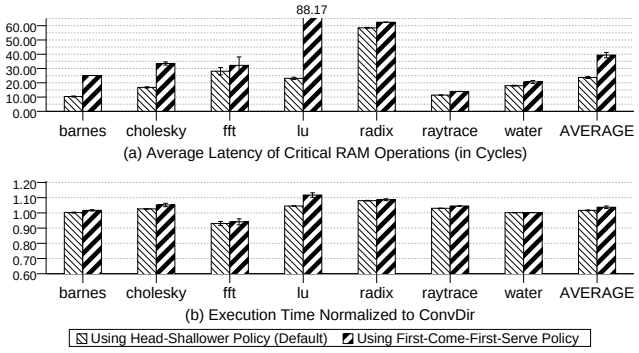

Figure 17: Performance Difference of *SpongeDirSmall* When Using Different Item Arrangement Policies

We can see that:

- The critical read latency is 2× faster when using a *Latency-Optimized* design.
- Despite the faster read operations, the overall performance is largely unchanged.
- The *Latency-Optimized* design consumes substantially more power - about 12.5× more energy. This is actually 11.82% of overall system power (23.84% for `radix`), an unacceptably large percentage.

Therefore, the *Energy-Optimized* design provides the appropriate trade-off between performance and energy consumption.

5.7. Sensitivity to Item Arrangement Policies

In Section 3.2, we described the default arrangement policy of SpongeDirectory—always placing the head items in shallower levels than body items. This is the *Head-Shallower* item arrangement policy. In order to evaluate the savings from this policy, we compare it to a *First-Come-First-Serve* policy which does not differentiate between head and body items. When inserting a new item, the *First-Come-First-Serve* policy puts it into the shallowest unused memristor level.

As is shown in Figure 17, graphing the comparison for *SpongeDirSmall* (the only configuration exercising deep levels):

- *First-Come-First-Serve* placement policy results in, on average, a 1.66x longer critical RAM operation latency, and, for benchmark `lu`, 3.81x.
- The latency difference in `lu` is enough to result in a 6.9% slowdown. The average slowdown was relatively small.

The energy consumption is virtually the same because writes consume most of the energy, and the policy does not affect the energy consumption of write operations.

This shows two important results. First, while based on sound design principles, *Head-Shallower* reduces critical read latency significantly, but it does not reduce overall execution time. Second, this is the second example that shows that read latency is not as critical as it could be, which is a very

important result given the emerging nature of memristors.

5.8. Sensitivity Analysis Discussion

Both aspects of our sensitivity analysis showed that, in general, our design is robust to difference in the over read latency. This is due to a combination of the small number of directory operations (Figure 4) and the long latency of the entire operation (Figure 3). This results in a simpler, more energy-efficient design because the allocation algorithm and memristor energy were in direct opposition with minimizing read latency.

This is important when projecting how our design will fair once memristors become a mature technology. We have intentionally chosen conservative estimates on latency and power based as much on current experiments as possible. It is likely that, as the technology matures, future designs will result in lower latency and energy than we have projected. We have shown, however, that our design is tolerant to differences in the latency. The energy is a much more critical design constraint.

5.9. Scaling SpongeDirectory beyond 256 cores.

As the system scales beyond 256 cores, the sharer vector entry will no longer fit in a single block. With the same provisioning rate and organization, there are two challenges. First, the maximum number of sharers has been scaled, so entries with many sharers will take more space. Second, the total number of entries that can map to the same directory entry has increased, increasing the effect of pathologically bad behavior.

We can integrate SpongeDirectory with other proposed directory compression techniques (see Section 6) to obtain higher scalability. For example, we can use a technique similar to SCD [29] to store sharer vectors hierarchically, depending on the number and location of sharers. For example, with 32,768 sharers, we could use a hierarchy with one head entry, up to 32 level-2 entries, and up to 32x32 leaf entries.

If this is not sufficient, we can combine the SCD and memristors scheme further. We would use multi-level memristors within the set, but once those are full, we would use the SCD scheme to "borrow" a block from another set.

6. Related Work

6.1. Multi-level Non-Volatile Memory

Recently, several projects have focused on architectural support for multi-level non-volatile memories (multi-level NVMs) from different angles. Qureshi et al. [25] improved the access latency of multi-level PCM by using a hardware-software hybrid scheme, which converts a multi-level PCM page into two single-level PCM pages when used often. Several projects [18][19] [24][33] focus on improving the MLC NVM write performance/energy/endurance by improving the NVM infrastructure and memory controller. Jiang et al. [17] also worked on improving the performance of a multi-level STT-RAM. Saadeldeen et al. [27] use memristors for branch prediction. However, to our knowledge, no previous work has used multi-level NVM to solve the coherence directory scaling problem.

6.2. Coherence Directory

Many projects have attempted to reduce the storage of on-chip directories as well as tolerate the variability in sharers and entries. The SpongeDirectory provides extra storage at the expense of higher latency and energy, whereas these schemes reduce storage needs or use those bits more efficiently. Most of these schemes could be combined with SpongeDirectory to provide even more area savings.

Building hierarchical directories [32][15] [1][20][22] is another way to reduce directory storage while still preserving exact sharer information. Martin et al. proposed a hierarchical solution for the in-cache directory [22] which embeds coherence information into a hierarchy of inclusive caches. They show that such a approach is efficient in terms of area, network traffic and energy. However, such hierarchical designs create complexity challenges. Another scheme, *Waypoint* [20], uses small directory caches on chip, overflowing extra directory entries to a special part of cacheable user-space memory. It requires over substantially higher directory lookup latency when there is an on-chip directory miss.

The Cuckoo Directory [12] was partly motivated by the observation that set-level non-uniform accesses to directory entries could induce an excessive number of invalidations. This uses a complex hashing technique which achieves almost the same invalidation rate as a fully associative sparse directory with only moderate (about 1.5×) overprovisioning. However, this scheme does not seek to reduce the sharer storage within a directory entry, thus it is not scalable in storage for many-core systems.

The SpongeDirectory is inspired by *Scalable Coherence Directory (SCD)* [29] which introduced a pointer-vector hybrid scheme to encode sharer information. SCD "borrows" blocks from underutilized sets, whereas the SpongeDirectory uses technology to provide extra space within the same set. SCD relies on high directory associativity, like Cuckoo Directory (Zcache [28]). Such schemes are complimentary to SpongeDirectory and could be used to provide further space savings.

Compression has been used by storing some information at the page level[8][26], using dual-grained tracking[4], and using many granularities[40][10]. Others have used varying compression schemes [30, 42, 41, 43].

7. Conclusions

In order to scale up coherence directories for future extra-scale many-core system, we propose SpongeDirectory, a sparse directory scheme utilizing multi-level memristor RAMs. Each SpongeDirectory block is able to dynamically

change its number of levels (thus total storage), according to current dynamic requirement.

Evaluations on a 256-core extreme-scale processor show that a SpongeDirectory optimized for low energy consumption has the performance of a conventional sparse directory with over 20× the storage space while using less energy.

Finally, SpongeDirectory uses technology to accommodate variation in directory demands. This could be combined with other schemes to reduce overall storage requirements such as using organization (i.e. SCD) or compression.

References

[1] M. E. Acacio *et al.*, "A two-level directory architecture for highly scalable cc-numa multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 67–79, 2005.

[2] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th IEEE International Symposium on High-Performance Computer Architecture*, 2003, pp. 7–18.

[3] F. Alibart *et al.*, "High-precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *CoRR*, vol. abs/1110.1393, 2011.

[4] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 341–350.

[5] M. Bach *et al.*, "Analyzing parallel programs with pin," *IEEE Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[6] R. J. Baker, *CMOS: circuit design, layout, and simulation*. Wiley-IEEE Press, 2011, vol. 18.

[7] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, pp. 16–29, Mar.–Apr. 2010.

[8] B. A. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th International Symposium on Computer Architecture*, 2011, pp. 93–104.

[9] X. Dong *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[10] L. Fang *et al.*, "Building expressive, area-efficient coherence directories," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013, pp. 299–308.

[11] L. Fang *et al.*, "Building expressive, area-efficient coherence directories," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 299–308.

[12] M. Ferdman *et al.*, "Cuckoo directory: A scalable directory for many-core systems," in *17th IEEE International Symposium on High-Performance Computer Architecture*, 2011, pp. 169–180.

[13] L. Gao *et al.*, "Digital-to-analog and analog-to-digital conversion with metal oxide memristors for ultra-low power computing," in *IEEE/ACM International Symposium on Nanoscale Architectures*, 2013, pp. 19–22.

[14] G. Grohoski, "Niagara-2: A highly threaded server-on-a-chip," in *Hot Chips 20*, 2008.

[15] S.-L. Guo *et al.*, "Hierarchical cache directory for CMP," *Journal of Computer Science and Technology*, vol. 25, pp. 246–256, Mar. 2010.

[16] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *19th International Conference on Parallel Processing*, 1990, pp. 312–321.

[17] L. Jiang *et al.*, "Constructing large and fast multi-level cell stt-mram based cache for embedded processors," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012, pp. 907–912.

[18] L. Jiang *et al.*, "Improving write operations in mlc phase change memory," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–10.

[19] M. Joshi, W. Zhang, and T. Li, "Mercury: A fast and energy-efficient multi-level cell based phase change memory system," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 345–356.

[20] J. H. Kelm *et al.*, "WAYPOINT: Scaling coherence to thousand-core architectures," in *19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 99–110.

[21] J. Li *et al.*, "A novel reconfigurable sensing scheme for variable level storage in phase change memory," in *Memory Workshop (IMW), 2011 3rd IEEE International*, 2011, pp. 1–4.

[22] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, pp. 78–89, Jul. 2012.

[23] J. E. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *16th IEEE International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[24] D. Niu *et al.*, "Low power multi-level-cell resistive memory design with incomplete data mapping," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 131–137.

[25] M. K. Qureshi *et al.*, "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *37th International Symposium on Computer Architecture*, 2010, pp. 153–162.

[26] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 241–252.

[27] H. Saadeldeen *et al.*, "Memristors for neural branch prediction: a case study in strict latency and write endurance challenges," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013, p. 26.

[28] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 187–198.

[29] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th IEEE International Symposium on High-Performance Computer Architecture*, 2012, pp. 1–12.

[30] R. Simoni, "Cache coherence directories for scalable multiprocessors," Stanford University, Technical Report CSL-TR-92-550, Oct. 1992.

[31] A. C. Torrezan *et al.*, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, vol. 22, no. 48, 2011.

[32] D. A. Wallach, "PHD: a hierarchical cache coherent protocol," Master's thesis, Massachusetts Institute of Technology, 1992.

[33] J. Wang *et al.*, "Energy-efficient multi-level cell phase-change memory system with data encoding," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, 2011, pp. 175–182.

[34] S. C. Woo *et al.*, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd International Symposium on Computer Architecture*, 1995, pp. 24–36.

[35] Q. Xia *et al.*, "Memristor-CMOS Hybrid Integrated Circuits for Reconfigurable Logic," *Nano letters*, vol. 9, no. 10, pp. 3640–3645, 2009.

[36] C. Xu *et al.*, "Design implications of memristor-based rram cross-point structures." in *DATE*, 2011, pp. 734–739.

[37] C. Xu *et al.*, "Understanding the trade-offs in multi-level cell reram memory design," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–6.

[38] J. J. Yang *et al.*, "Memristive switching mechanism for metal//oxide//metal nanodevices," *Nature Nanotechnology*, vol. 3, pp. 429–433, 2008.

[39] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, pp. 13–24, 2013.

[40] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directory," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2013)*, no. EPFL-CONF-195669, 2013.

[41] J. Zebchuk *et al.*, "A tagless coherence directory," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 423–434.

[42] H. Zhao *et al.*, "SPATL: Honey, i shrunk the coherence directory," in *20th International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 33–44.

[43] H. Zhao, A. Shriraman, and S. Dwarkadas, "SPACE: Sharing pattern-based directory coherence for multicore scalability," in *19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 135–146.