International Journal of Engineering Sciences Paradigms and Researches (Volume 47, Issue: Special Issue of January 2018)
ISSN (Online): 2319-6564 and Website: www.ijesonline.com

SpongeDirectory: Number of co Memristors and Flexible Sparse Directories

Dr. B.Purna Satyanarayana[1], Dr. Chinmay R. Pattanaik[2]*
[1] Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Associate Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
bpurnasatyanarayana@thenalanda.com, chinmayaranjan@thenalanda.com*

ABSTRACT

In many-core systems, cache-coherent shared memory is essential for programmability. Many directory-based methods have been put forth, however due to dynamic, non-uniform sharing, they are always energy, performance, or storage space inefficient.

A sparse directory structure that makes use of multi-level memristory technology is called SpongeDirectory, which we introduce. By expanding the number of bits held on a single memristor device, SpongeDirectory grows directory storage in-place when necessary, exchanging latency and energy for storage. We investigate several SpongeDirectory setups and discover that the most competitive option uses memristors with a provisioning rate of 0.5x and is designed to use little energy. With 18 percent less storage space and 8 percent reduced energy usage, this ideal SpongeDirectory setup performs on par with a traditional sparse directory.

Categories and Subject Descriptors

B.3.2 [Memory Structures]: Design Styles; B.7.1 [Integrated Circuits]: Types and Design Styles

Keywords

Sparse Directories; Multi-Level Memristors

1. INTRODUCTION

As the number of processor cores increases, multi-/many-core chips tend to favor directory- over bus-based snooping coherence due to bandwidth needs. Simple directory schemes based on sharer vectors are not scalable, however, because storage increases super-linearly with the number of cores. A scalable, efficient directory scheme is needed for future extreme-scale many-core system. Many directory schemes (e.g., [14][42][44][20][30]) have been proposed to reduce the storage requirements at the cost of energy, latency or complexity. For example, duplicate-tag [14] and tagless [42] directories provide scalable performance and area, but they are not energy efficient. The recently proposed SCD [30] scheme is scalable, but only if it is coupled with a complex ZCache architecture[29].

Requirements for directory coherence storage are quite different among different applications. Some applications have a large number of directory entries, but most directory entries have very few sharers, whereas others have fewer directory entries, with more sharers per entry. Using conventional SRAMs, it is quite complex to provide the flexibility for both situations while keeping overall storage overhead low [11][29].

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 [36], 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 requirements of different directory storage requirements in different situations. The extra memristors may be used either for more sharers in an entry or for more entries in a set, allowing for flexible needs within and across applications.

This paper makes the following contributions:

- Provide a model projected from empirical data for latency and energy requirements for memristors.
- Introduce an encoding scheme that requires only one memristor bit to store the number of levels being used.
- Design and evaluate a coherence directory with memristors that, with the optimized configuration, saves 18x storage, results in negligible degeneration on overall system performance, and consumes 8× less energy than 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 in Section 7.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628081.

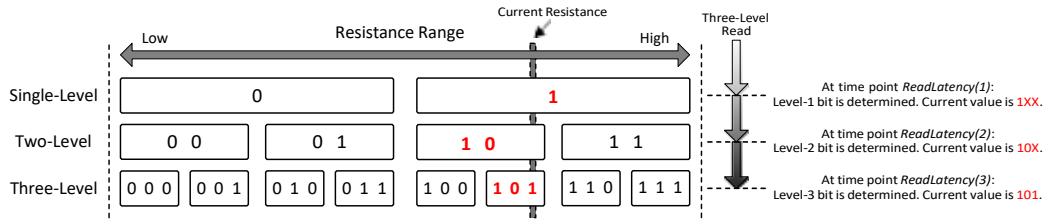

Figure 1: Multi-level memristor cell, and the step-wise manner of a multi-level memristor read. The same resistance can be read as different numbers of bits, and the most significant bits are read first.

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.

Multi-Level Memristor Storage vs Latency

Memristors [39][40] 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 (at the expense of energy and latency).

A memristor cell [39] 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][38]. Figure 1 shows that the same resistance can be interpreted as 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 comes at a cost in latency and energy. Here we describe in detail the characteristics of multi-level memristor operations:

Multi-level Memristor Reads

As the number of bits being stored in a single memristor increases, the ADC must distinguish between 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.

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 very high latency (up to hundreds or even thousands of nanoseconds). Fortunately, the operation can be interrupted between iterations to service more critical operations (e.g. a read).

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.

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][30][9][4][11]) work on improving the scalability 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 proposal.s In addition, it as 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 (address of corresponding memory block), state (three states in a conventional MESI protocol) and sharers (which cores' caches currently contain the memory block).

If a processor core asks for a 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 disadvantage of the original sparse directory is its storage requirement, since it must hold all information perfectly. Depending on who is sharing what, there are two sources of storage inefficiency:

- Storing sharer information in an entry: In a sharer-vector form, the storage of such information is linear with the number of cores and thus not scalable.
- Slice-level and set-level non-uniformity: 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 [8]) 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]).

In this paper we analyze 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 coherence directory using multi-level memristors. This analysis is analyzes both spatial and temporal behavior.

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.*

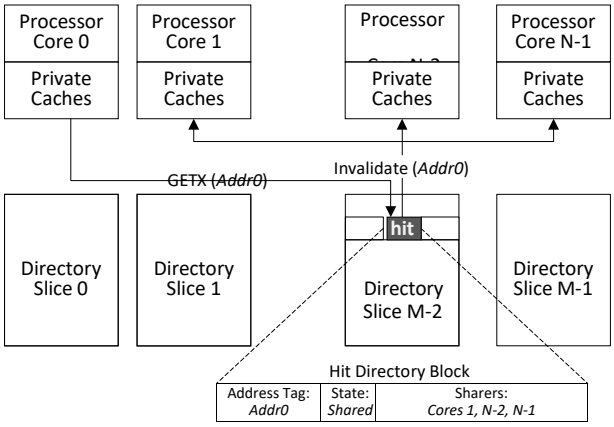

Figure 2: An example of a sparse directory scheme used in a multi-/many-core processor to track private cache coherence information. A sparse directory uses set-associative cache-like structures and is is distributed into slices (M slices in this figure).

- 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 1144× and 24.9× 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.*

Directory Design: Spatial

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., barnes and raytrace) have significantly 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.

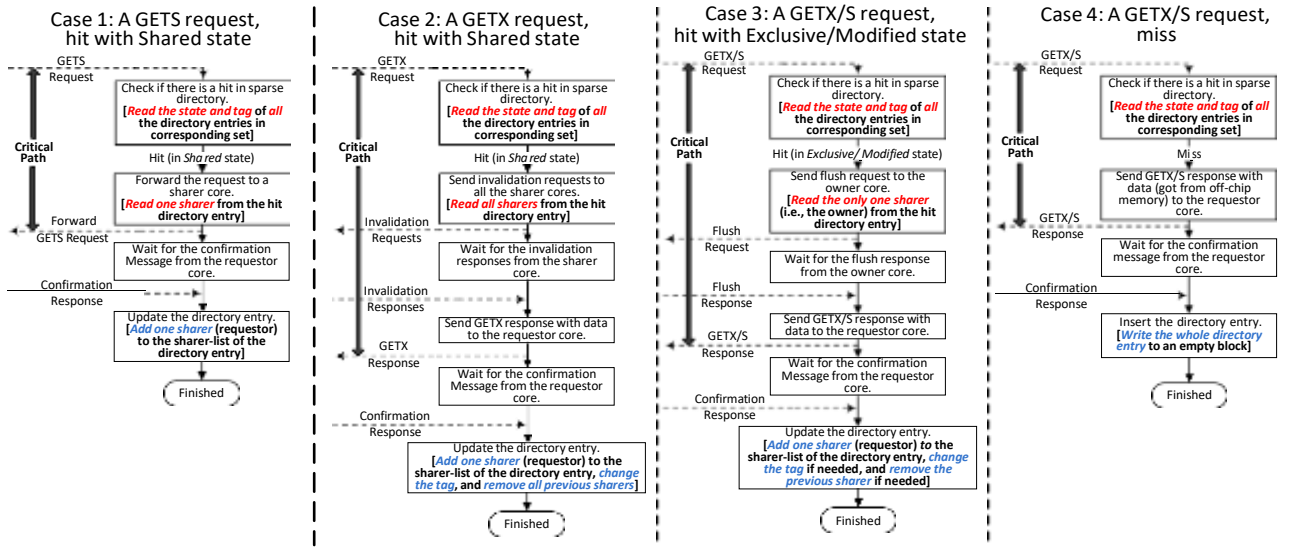

Figure 3: Behaviors of GETX and GETS requests in a conventional sparse directory using MESI protocol showing reads (red) and writes (blue). Goal: reduce latency for common operations on critical path. We see that: (1) Only reads are on the critical path. (2) All reads require state and tag, and most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

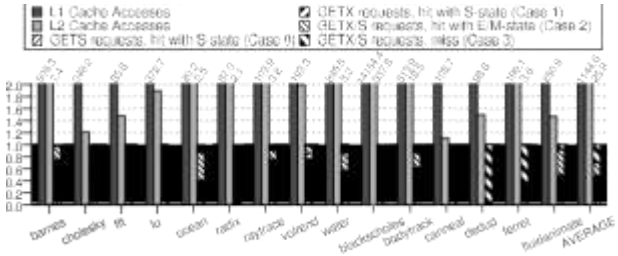

Figure 4: Number and type of memory accesses normalized to total directory accesses. Directory accesses are relatively infrequent, therefore latency tolerant.

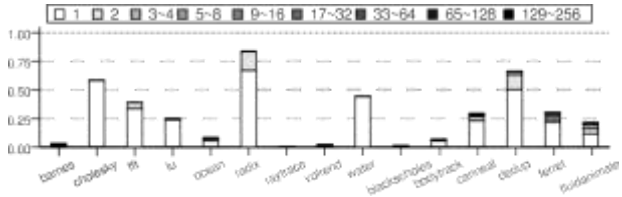

Figure 5: Number of directory entries with X sharers, normalized to the number of cache blocks showing variance in need for entries (radix) and sharers (raytrace).

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, like a normal sparse directory slice, is an N-way set-associative memory. Each set has N *blocks* in it. The difference is that the memory is made of multi-level memristor RAMs, therefore each *block* can be configured individually with a number of levels. We call each level of a SpongeDirectory *block* a SpongeDirectory *item*.

Identifying the Level of a 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.

Each SpongeDirectory *block* has one *usage bit*. If the correspondent *item* is the deepest valid *item* of the block (e.g., the Level 3 *item* in Figure 1), its *usage bit* is 0; otherwise, it is 1. Therefore, A multi-level memristor read can stop once it reads a *usage bit* with value 0.

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 more cores share the cacheline, *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 per sharer, but requires space for all cores (not just those sharing it). This is efficient for many sharers. Similar to prior work[30], each entry may be configured as either

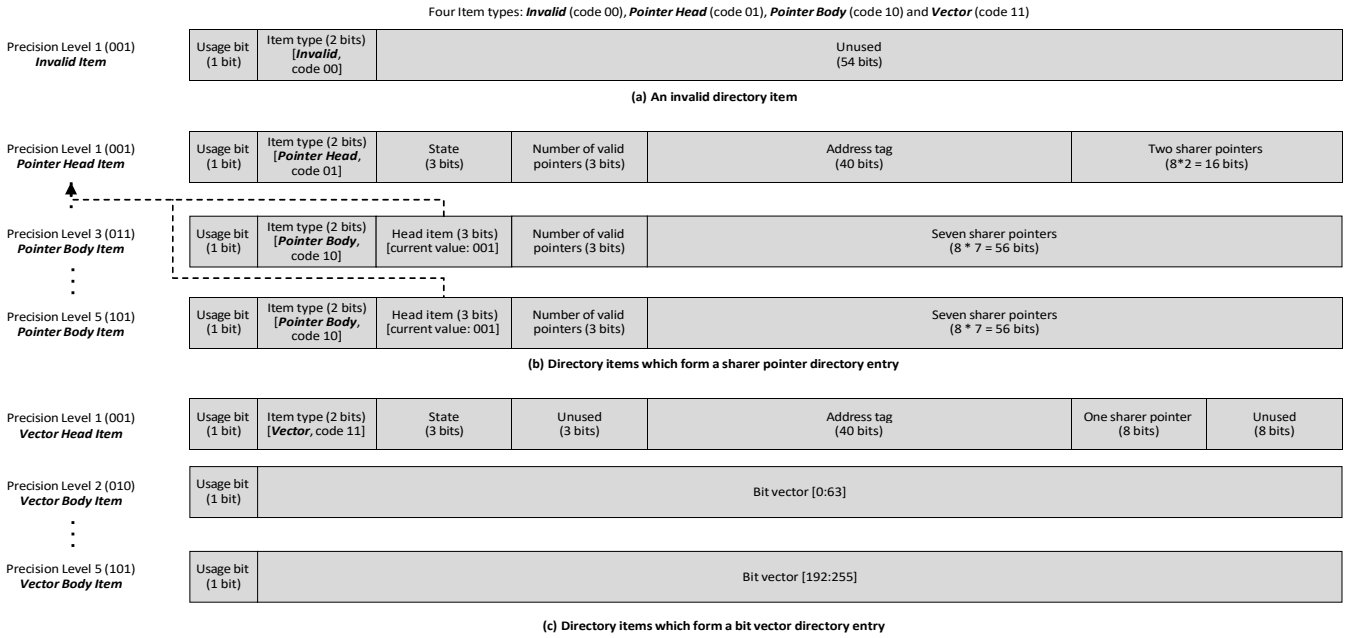

Figure 6: The entry formats of SpongeDirectory

scheme to obtain the advantage of both *sharer pointer* and *sharer vector* schemes.

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 two sharer pointers. 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 four *items*). With its *head item*, a *sharer vector directory entry* requires all five *items* of a *block*. To ensure fast directory responses, a *sharer vector head item* preserves a sharer pointer.

Figure 7 shows how items would be stored in the same block. We add a few restrictions to reduce read latency:

- A single *block* may contain several *directory entries* in sharer pointer format.
- One *block* can hold up to five *items* (out of a possible seven [3]).
- *Head items* are stored in the shallowest levels for fast completions of cases 1, 3, and 4 in Figure 3.

We also add restrictions to simplify the design:

- All *items* for the same *directory entry* must reside in the same block to simplify accesses.
- A *directory entry* in *sharer vector* format uses the entire *block*.

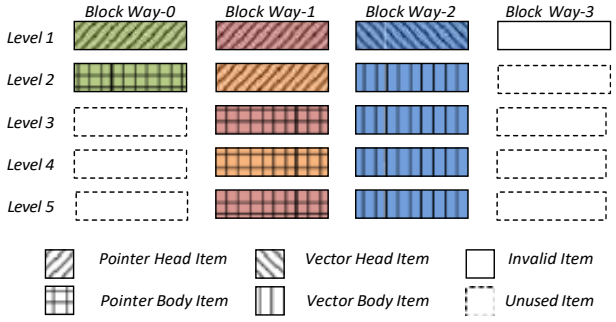

Figure 7: A snapshot of a 4-way SpongeDirectory set. *Items* with the same color belong to the same directory *entry*.

Minimizing Write Energy

As will be shown in Section 4.2, 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* that cross the threshold of the number sharers in an item in *sharer pointer format* requires the entire *block* to be 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* may leave a hole, so a sharer pointer may need to be moved. Therefore, the maximum is two *sharer pointers* and the *pointer count field.*.
- Adding a sharer in *sharer vector format* requires modification of only a single bit.
- Removing a sharer in *sharer vector format* modifies only a single bit unless the removed sharer is also in

the *sharer pointer* field of the *head item*. In this case, we also need to modify the *sharer pointer* field.

- Changing directory entry format requires the entire block to be written (see below).

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. All other *directory entries* sharing the same *block* are evicted, sharer pointers are translated to vector locations, and *sharer vector items* are written to 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. The directory could toggle across the threshold, so we do not downgrade at this time. 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, so we only downgrade a *sharer vector directory entry* at this time.

Buffering Recently Finished Requests

There are still two challenges for SpongeDirectory scheme in some applications: limited write durability and long write latency. As we will show later, if a very small number of blocks are written many times, as in `fluidanimate`, this will hurt the lifetime of that block, as well as suffer delays due to the long write latencies. Therefore, we propose a buffer that holds the most recently finished directory requests to capture subsequent operations to those entries.

4. METHODOLOGY

In this section, we describe the two main aspects to the experiments. First, we describe the different configurations we evaluated. Second, we describe the simulations and models we used to obtain our numerical results.

Directory Configurations

Our baseline is a conventional sparse directory architecture. In addition, we calculate the storage requirements of the SCD architecture. For the SpongeDirectory configurations, we have several configurations that vary the size, memristor type, organization, and buffering in order to evaluate the effectiveness of our design decisions. Table 2 summarizes the storage requirements of the three basic sizes. All of the variations only change one attribute as compared with *SpongeDirMid*, so we describe only the difference between SpongeDirMid and that configuration.

- ConvDir is a conventional sparse directory architecture using a sharer-vector scheme with a 2× provisioning rate.
- SpongeDirMid uses energy-optimized memristors with a 0.5× provisioning rate and an 8-entry buffer of the most recently completed requests.
- SpongeDirSmall uses a 0.25× provisioning rate.
- SpongeDirLarge uses a 1× provisioning rate.
- SpongeDirMid-NoBuffer has no buffer of the most recently completed requests.
- SpongeDirMid-FastMemristor uses latency optimized memristors.
- SpongeDirMid-6Levels uses a narrower format requiring 57 bits per block instead of 65 bits — each *Pointer Head Item* has one sharer pointer instead of two. This will utilize deeper levels more often.
- SpongeDirMid-SimplePolicy is SpongeDirMid but does not take advantage of the organization of items within a block. Although head items are stored in the shallowest levels, this reads the entire entry before processing the request.

Models

For each configuration, we calculate the storage requirements as well as the read latency, write latency, and energy. We developed models for memristors projected from recent experimental data. We used these values as inputs to our simulation infrastructure.

Simulation.

We implement our simulation platform with the multi-threaded 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 512KB 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 design using nine SPLASH-2 benchmarks [35] (barnes, cholesky, fft, lu, ocean, radix, raytrace, volrend and water) and six PARSEC benchmarks [7] (blackscholes, bodytrack, canneal, dedup, ferret and fluidanimate). As shown in Figures 4 & 5, these benchmarks exhibit a wide variety of directory behaviors.

To reduce simulation time, Graphite [23] uses relaxed coordination among the threads, and the developers show that the simulation inaccuracy caused by the approach is acceptable. However, this means that the requests of different threads sometimes arrive to a directory slice not in a time order, making it impossible to model the queuing delay of different operations. As a result, we first present the execution time without considering the queuing delay. We then separately present the memristor operation time in the busiest directory slice to show when queuing delay is likely to considerably prolong the execution time.

Storage.

Table 2 shows the storage requirements of the directory schemes, which is affected by two parameters: (1)storage of each *directory block* ; (2)*provisioning rate* of the sparse directory scheme, defined as the total number of *directory blocks* divided by the total number of cache blocks in all private caches.

For the first parameter, due to their multiple storage formats, both SCD and SpongeDirectory require smaller *directory blocks* than a conventional sparse directory.

The *provisioning rate* of a conventional sparse directory is 2×. Because SCD mimics a much larger associativity with its Zcache-like infrastructure, it only needs a provisioning rate of 1×. Each *SpongeDirectory block* can be configured to hold multiple *directory entries*, further

Table 1: Simulation Parameters

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, parallel access, 2-cycle latency, 0.82pJ tag lookup, 88.71pJ data access.
L2 Caches	private, 512KB per processor, 8-way, sequential access, 11-cycle latency, 4.62pJ tag access, 74.70pJ 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. ConvDir: RAM read/write latency: 1 cycle (1ns); buffering 8 most recent requests. SpongeDirectories: by default, buffering 8 most recent requests, using energy-optimized memristors, head item shallower policy and 5-level memristors, a sharer pointer format directory entry having at most three *items*.
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	Block organization	Prov. rate	Directory RAM storage per tile (caches have 624.375 KB RAM per tile)
SpongeDirSmall	SpongeDirectory	memristor	4-way, 512 sets	0.25×	65 bits per block (5 levels/bit) -> ~16KB (2.56% cache)
SpongeDirMid (default)	SpongeDirectory	memristor	4-way, 1024 sets	0.5×	65 bits per block (5 levels/bit) -> ~32KB (5.13% cache)
SpongeDirLarge	SpongeDirectory	memristor	4-way, 2048 sets	1.0×	65 bits per block (5 levels/bit) -> ~64KB (10.25% cache)
SpongeDirMid -6Levels	SpongeDirectory	memristor	4-way, 1024 sets	0.5×	57 bits per block (6 levels/bit) -> ~28KB (4.48% cache)
ConvDir	Conventional Sparse Directory	SRAM	4-way, 4096 sets	2.0×	307 bits / block: 3-bit state, 40-bit tag, 8-bit sharer pointer, 256-bit vector -> 614 KB per tile (98.3% of cache).
SCD	SCD [30]	SRAM	Zcache [29], 4 ways, 2048 sets	1.0×	71 bits per block: 3-bit state, 40-bit tag, 2-bit type, 26-bit sharer info field -> 71 KB per tile (11.37% of cache).

Table 3: Energy and Latencies 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	16KB	memristor	1.651ns	4.763ns	41.464pJ	3.172pJ	11000.6um^2	3563.8um^2
SpongeDirMid	32KB	memristor	1.698ns	5.463ns	41.775pJ	4.008pJ	11464.8um^2	6363.0um^2
SpongeDirLarge	64KB	memristor	1.809ns	6.739ns	42.397pJ	4.167pJ	12411.4um^2	8011.9um^2
SpongeDirMid -6Levels	28KB	memristor	1.670ns	5.439ns	36.582pJ	3.341pJ	10404.7um^2	5671.0um^2
ConvDir	614KB	SRAM-based 4-way cache	< 1.000ns		tag: 11.58pJ data: 74.70pJ		968655um^2	

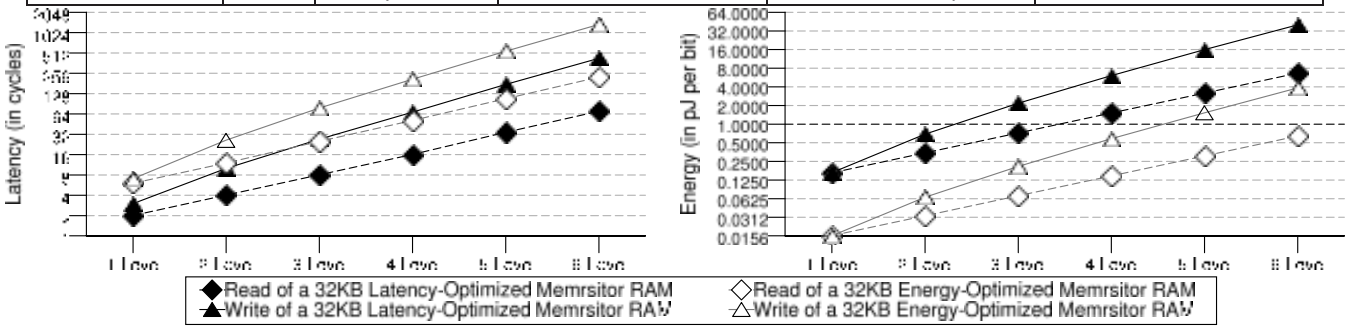

Figure 8: The latency and energy consumption of 32KB multi-level memristor RAMs used in *SpongeDirMid*.

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

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

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

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

reducing its provisioning rate. We evaluated three sizes of *SpongeDirectories* — 0.25×, 0.5×, and 1×.

In conclusion, *SpongeDirectory* has smallest because it utilizes both a compact *directory block* format as well as a low *provisioning rate*. This results in *SpongeDirMid* (our default SpongeDirectory configuration) requiring 1/18th of the storage of a conventional sparse directory and less than half of SCD.

Memristor and SRAM-Cache Modeling.

For the detailed modeling of memristors, we use nvsim [10] 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 [37] 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 twice as fast as the *Energy-Optimized* configuration, but they consume almost 10x the energy and require more area. In addition, we see that for the *Energy-Optimized* configuration, all SpongeDirectory configurations consume substantially less energy than a conventional directory when accessing the lowest level.

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-level memristor technique.

As 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, the number of iterations grows linearly with the number of write levels. Equation 2 reflects this curve. *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 [32], 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 the read operation is constant (because the read voltage is constant), we obtain 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]. Memristor write pulses can be less than one nanosecond [32], therefore we assign $WritePulseLatency = 1ns$ in Equation 2. Figure 8 shows the latency and energy consumption of multi-level operations on 32KB memristor RAMs (with Latency-Optimized or Energy-Optimized configurations).

5. RESULTS

In this section, we present the evaluation of SpongeDirectory in terms of eviction rate, critical directory access latency, overall performance, directory energy consumption and lifetime. For clarity, in each graph, we remove any variations whose results are nearly identical to *SpongeDirMid*.

Eviction Rate

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

We can see from Figure 9 that, even with a provisioning rate of only 0.25×–1×, SpongeDirectory has a low eviction rate (on average all less than 1%). Evictions occur in SpongeDirectory for three reasons: format upgrades, non-uniform accesses, and lack of capacity.

When SpongeDirectory upgrades directory entries from the sharer pointer format to the sharer vector format, all other entries in the same block are evicted. These evictions could be reinstated at the cost of extra complexity for upgrades. However, given the fact that this occurs rarely in our experiments, there is little justification to introduce this complication.

Directory entries are sometimes non-uniformly distributed [12], leading to evictions in the hotspot directory sets. The SpongeDirectory gains much more associativity by increasing the number of levels it uses (at the cost of latency and energy consumption). For example, *SpongeDirSmall* can exhibit a lower eviction rate than *ConvDir* (`canneal` and `ferret`). As we will see in Section 5.3, this advantage sometimes helps SpongeDirectory achieve higher performance than a much larger conventional directory.

While extra levels provides more tolerance in a particular set than the conventional directory, the overall capacity of *SpongeDirSmall* is less than the conventional directory. 0.25× * 5 levels = 1.25×, which is still substantially smaller than conventional directory's 2×. Thus, the conventional directory can hold many more entries, as long as they are not concentrated in certain sets. This is illustrated in `radix`, where *SpongeDirSmall* experiences an eviction rate of 7.98% because of insufficient capacity in the hotspot *SpongeDirectory* slice.

Critical Read Operation Latency

As discussed in Section 2.2, the performance of a directory scheme is dependent on critical reads. In Figure 11, we present the latency of critical read accesses of different directory architectures. Compared to a conventional sparse directory, SpongeDirectories require many more cycles (on average 7.56 cycles for *SpongeDirSmall*, 4.44 cycles for *SpongeDirMid* and 4.39 cycles for *SpongeDirLarge*, compared with 0.58 cycles for *ConvDir*) to perform critical read accesses.

This long latency is due to two factors of the memristor technology. First, as an emerging technology, memristor devices still have much longer access latency compared with mature SRAM devices (Table 3). Second, multi-level memristor operations require more latency. Even with

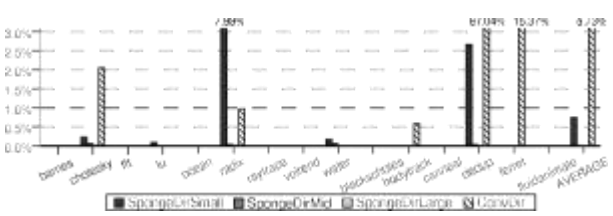

Figure 9: Directory Eviction Rate

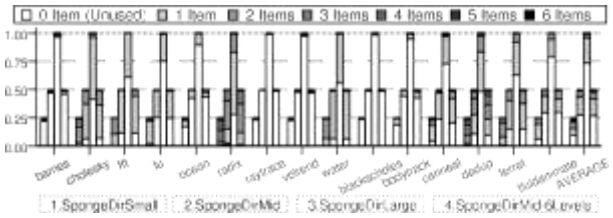

Figure 10: SpongeDirectory Blocks with Different Items, Normalized to Number of Cache Blocks

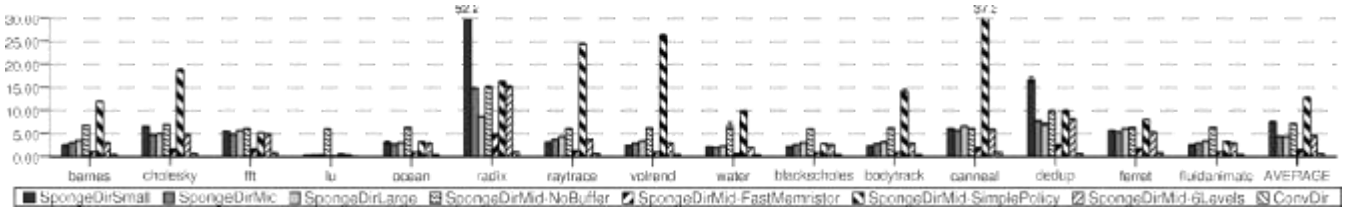

Figure 11: Average Latency of Critical Read Operations (cycles)

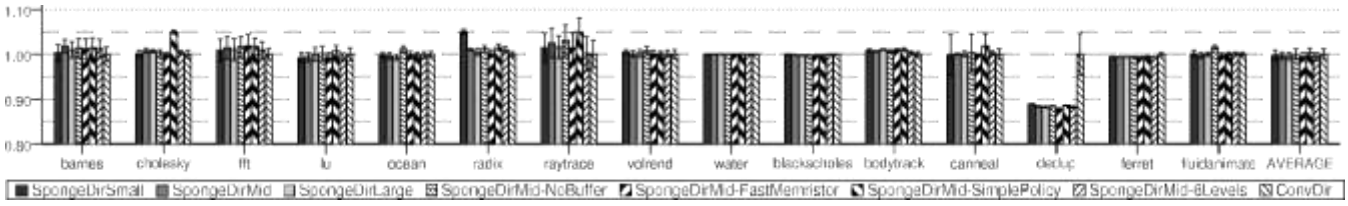

Figure 12: Normalized Execution Time (without considering directory request queuing)

careful design to reduce accesses to deeper levels, the lower the provisioning rate, the more often these deeper levels are used.

As shown in Figure 11, design choices such as provisioning rate, buffering, which memristors we use, and the organization policies, affect the critical read latency. Figure 10 shows the most direct reason for rising latency - increased accesses to deeper levels.

Provisioning Rate affects evictions and how often deep levels are acessed. Two benchmarks, radix and dedup, exhibit long critical latency (52.2 and 16.3 cycles respectively) for *SpongeDirSmall* due to increased accesses to deep levels.

Request Buffering helps reduce the critical latency because part of the requests are bypassed from accessing memristors. On average, the critical read latency of *SpongeDirMid-NoBuffer* is 2.71 cycles longer than *SpongeDirMid*.

Memristor Design can choose to optimize for either latency or energy. Optimizing for latency (*SpongeDirMid-FastMemristor*) is 2× faster (on average 1.47 cycles versus 4.44 cycles) than optimizing for energy (*SpongeDirMid*).

Item Organization Policies were chosen to increase the number of accesses to lower levels. Not utilizing this optimization (*SpongeDirMid-SimplePolicy*) can result in much longer critical read latency—for cholesky, 18.79 cycles compared with 4.81 cycles.

Narrower format also increases the access to deeper levels, as shown by *SpongeDirMid-6Levels*. While it did not greatly affect the critical read latency, it has a larger effect on the time when the memristor is busy (see Section 5.3).

Overall Performance

The overall performance of different schemes is determined by two factors — critical latency in the RAM discussed in Section 5.2 and the queuing time to wait for previous directory operations to finish. As discussed in Section 4, Graphite cannot model the second phenomena. Therefore, we first show the overall execution time not taking into account directory queuing time. We then show the percentage time the busiest slice was being used and discuss the effect of queuing time on overall performance.

Execution Time without Queueing Time.

Figure 12 provides a comparison of overall performance due to read latency 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, more compact format (resulting in 18x fewer bits) and longer memristor access latency.

Closer inspection of the eviction rates and levels used within SpongeDirectory explain the variance in the results. Benchmark dedup has a much lower eviction rate in the SpongeDirectory than the conventional directory, and, although higher levels are quite often used, all SpongeDirectory configurations experience a 10% performance gain.

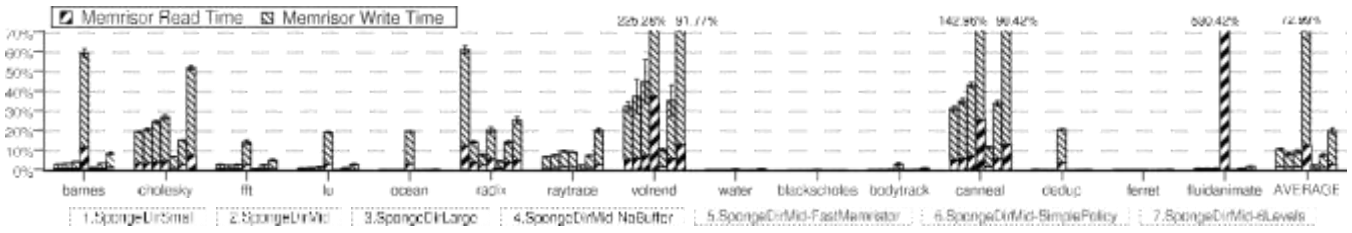

Figure 13: Percentage time busiest SpongeDirectory slice is serving requests

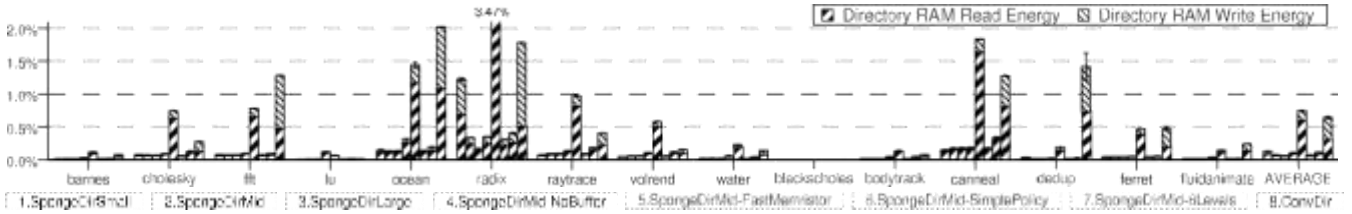

Figure 14: Percentage of on-chip memory dynamic energy

We observe two cases of obvious slowdown of SpongeDirectories, and we look for the detailed reasons:

- `cholesky` on *SpongeDirMid-SimplePolicy* : `cholesky` needs much longer critical read latency in the *SpongeDirMid-SimplePolicy* than all other directory architectures. This indicates that careful item organization of the SpongeDirectory is indeed necessary.
- `radix` on *SpongeDirSmall* : `radix` suffers from both a high eviction rate and long critical read latency due to accesses in deep levels. Therefore, we need more capacity than the 0.25× provisioning rate of *SpongeDirSmall* to avoid such pathologically bad behavior.

Considering Queueing Time.

With the help of suspendable multi-level memristor writes, long-latency non-critical memristor accesses typically do not slow down critical directory operations. However, this is based on the assumption 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.

Figure 13 shows the percentage of memristor access time in the busiest SpongeDirectory slice. For *SpongeDirMid*, the average value is 8.37% (up to 35.1% when running `canneal`), which is acceptable. However, several other *SpongeDirectory* design choices sometimes cause unacceptably long memristor operation time :

- No request buffering leads to memristor operation time in the busiest slice which is longer than total exection time for `fluidanimate`, `volrend`, and `canneal`. This would result in a slowdown of at least 1.5x-5.3x.
- Using six levels instead of five levels leads to memristor operation time in the busiest slice which is almost same with total exection time for `canneal` and `volrend`, likely slowing down those two applications.
- Small provisioning rate (*SpongeDirSmall*) leads to high occupancy in the busiest slice of `radix` - 61.45% of overall execution time.

When we consider the overall execution time and time the busiest slice was being utilized, the reasons for our default parameters are clear. Although *SpongeDirMid-FastMemristor* individual operations are considerably faster than with *SpongeDirMid*, this advantage is not reflected in terms of overall execution, nor an unacceptably large percentage time the the busiest directory slice was being used. Therefore, we do not need to use latency-optimized memristors to maintain competitive performance. In addition, it is clear that buffering is critical. *SpingeDirMid* provides the best trade-off between a low eviction rate, small storage, and good performance. Finally, a wider format that uses only 5 levels is worth the small extra area.

Dynamic Directory Energy Consumption

Figure 14 shows the breakdown of dynamic energy consumption of different directory schemes. We see that, on average, all of the SpongeDirectories configurations with energy-optimized memristors consume less energy than a conventional directory. *SpongeDirMid* consumes about half the energy of a conventional directory. Even *SpongeDirMid-FastMemristor*, which consumes by far the most energy, consumes only slightly more energy than a conventional directory.

If we compare the these configurations, we see that:

- It is clear that the performance benefit from latency-optimized memristors is far outweighted by the energy benefit of energy-optimized menristors.
- Request buffering further reduces the energy consumption. When executing `ocean`, *SpongeDirMid* consumes half the energy of *SpongeDirMid-NoBuffer*.
- A small provisioning rate can substantially increase the energy consumption. For example, when executing `radix`, *SpongeDirSmall* consumes nearly three times more energy than *SpongeDirMid* due to increased accesses to deeper levels.

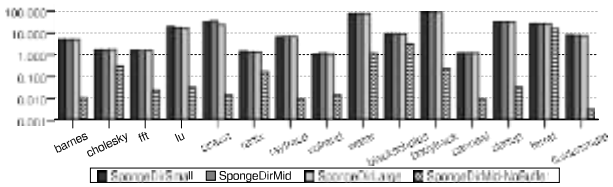

Figure 15: Worst Case Lifetime (in Years)

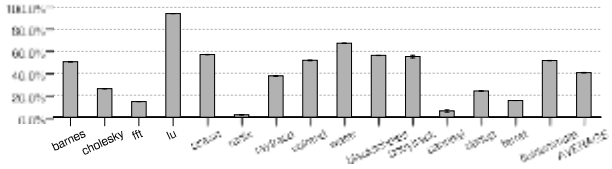

Figure 16: Percentage of Bypassed Memristor Accesses of SpongeDirMid due to Buffering

Memristor Lifetime

Based on statistics provided by Yang et al.'s work [40], we assume the endurance of each SpongeDirectory block is 10^{12} writes and report the lifetime of the most hotspot SpongeDirectory block in Figure 15. We can make two major conclusions. First, the provisioning rate does not have a large effect on worst-case lifetime. The worst-case lifetime of *SpongeDirMid* over all benchmarks is over 1 year. Since the hotspot SpongeDirectory block is unlikely to be the same block over long runs, we anticipate the overall lifetime of *SpongeDirMid* to be at least several years.

Second, request buffering is required for a feasable memristor implementation. Without buffering (*SpongeDirMid-NoBuffer*), in most cases the worst lifetime is below 1 year (down to 0.003 year in `fluidanimate`). This is because buffering is very effective at reducing the access rate of applications with high accesses to specific blocks when it is critical, as shown in Figure 16. This shows the percentage of bypassed memristor accesses with directory request buffering. For some benchmarks, a large percentage of directory requests are bypassed (up to 94.29% for `lu`); whereas for some other benchmarks, few directory requests are bypassed (down to 2.08% for `radix`). Note that, even though `radix` has a small bypass rate, request buffering still increases the worst case lifetime by 7.6 times. This is because the buffered requests are exactly the ones visiting hotspot *blocks*.

Summary

In summary, experimental results show that *SpongeDirMid* is a reasonable design choice. This design choice uses the following configurations for reasons:

- 0.5× provisioning rate provides a balance between good performance (compared to smaller) and storage space (compared to larger).
- Buffering of eight most recently finished requests dramatically increases the lifetime of the memristors and reduces the amount of time the memristors are busy (allowing for critical reads to interrupt long-latency write operations).
- Using energy-optimized memristors substantially reduces the total energy and area requirements with negligible reductions in overall performance.
- Using a head shallower item organization policy reduces overall execution time in some cases (e.g., `cholesky`).
- Using a wider, shallower 5-level memristor scheme instead of a narrower, 6-level scheme decreases the overall memristor operation time, allowing for more critical reads to interrupt long-latency write operations in some cases (e.g., `canneal`).

6. RELATED WORK

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][34] 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. Sampson et al. [28] propose to improve the performance, lifetime or density of multi-level PCM with an approximate storage technique.

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 [33][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×) over-provisioning. 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)* [30] 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 [29]). 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[9][26], using dual-grained tracking[4], and using many granularities[41][11]. Others have used varying compression schemes [31, 43, 42, 44].

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 18× the storage space while using 8× 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.

8. REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "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, L. Gao, B. Hoskins, and D. B. Strukov, "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, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil *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] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, pp. 16–29, Mar.–Apr. 2010.

[9] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th International Symposium on Computer Architecture*, 2011, pp. 93–104.

[10] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "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.

[11] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang, "Building expressive, area-efficient coherence directories," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013, pp. 299–308.

[12] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "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, F. Merrikh-Bayat, X. Guo, D. B. Strukov, and K.-T. Cheng, "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, H.-X. Wang, Y.-B. Xue, C.-M. Li, and D.-S. Wang, "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, B. Zhao, Y. Zhang, and J. Yang, "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, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "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, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAYPOINT: Scaling coherence to thousand-core architectures," in *19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 99–110.

[21] J. Li, C.-I. Wu, S. C. Lewis, J. Morrish, T.-Y. Wang, R. Jordan, T. Maffitt, M. Breitwisch, A. Schrott, R. Cheek *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, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *16th IEEE International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[24] D. Niu, Q. Zou, C. Xu, and Y. Xie, "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, M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis, "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, D. Franklin, G. Long, C. Hill, A. Browne, D. Strukov, T. Sherwood, and F. T. Chong, "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] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 25–36.

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

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

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

[32] A. C. Torrezan, J. P. Strachan, G. Medeiros-Ribeiro, and R. S. Williams, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, vol. 22, no. 48, 2011.

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

[34] J. Wang, X. Dong, G. Sun, D. Niu, and Y. Xie, "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.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd International Symposium on Computer Architecture*, 1995, pp. 24–36.

[36] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov, and Others, "Memristor-CMOS Hybrid Integrated Circuits for Reconfigurable Logic," *Nano letters*, vol. 9, no. 10, pp. 3640–3645, 2009.

[37] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, "Design implications of memristor-based rram cross-point structures." in *DATE*, 2011, pp. 734–739.

[38] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "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.

[39] J. J. Yang, M. D. Pickett, X. Li, O. A. A., D. R. Stewart, and R. S. Williams, "Memristive switching mechanism for metal//oxide//metal nanodevices," *Nature Nanotechnology*, vol. 3, pp. 429–433, 2008.

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

[41] 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.

[42] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 423–434.

[43] H. Zhao, S. Arrvindh, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, i shrunk the coherence directory," in *20th International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 33–44.

[44] 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.