Software-Hardware Managed Last-level Cache Allocation Scheme for Large-Scale NVRAM-based Multicores Executing Parallel Data Analytics Applications

Masab Ahmad*†, Halit Dogan*, Fabio Checconi†, Xinyu Que†, Daniele Buono†, Omer Khan*
*University of Connecticut, Storrs, CT, USA
†IBM Research, Yorktown Heights, NY, USA

Abstract—**Developments in machine learning and graph analytics have seen these fields establish themselves as pervasive in a wide range of applications. Non-volatile memory (NVRAM) offers higher capacity and information retainment in case of power loss, therefore it is expected to be adopted for such applications. However, the asymmetric access latencies of NVRAM greatly degrade performance. The focus of this paper is to reduce the effect of memory access latency on emerging machine learning and graph workloads. The proposed mechanism uses software tagging of application data structures so as to control on-chip cache evictions based on data type and reuse patterns in an NVRAM based multicore system. Learner models are developed that are capable of predicting cache allocations for a variety of machine learning and graph applications. The optimized learning model yields an average performance benefit of 21% compared to a system that does not optimize for the write latency challenges in NVRAM.**

I. Introduction

Machine learning and graph analytics are becoming the epitome of data processing applications. Algorithms stemming from such applications crunch on large data sizes. One of the most revolutionary changes to cater for this data requirement is the advent of non-volatile memory (NVRAM) as the main memory replacement for DRAMs [27]. Such memory is power-persistent, it's structuring allows larger storage capacities [24], and is a cheaper per-byte alternative to DRAM [29]. Large-scale multicore machines with NVRAM are expected to execute workloads that exploit plentiful parallelism [32]. NVRAM-only main memory systems are utilized because machine learning and graph applications can lookup crucial training and intermediate data even when the device is turned off. This is analogous to mobile and car applications, where devices are turned off frequently. However, the heterogeneous latencies of NVRAM lead to many performance challenges. Higher write latencies are associated with NVRAM because entire dirty pages need to be re-written in multiple segments of the main memory [32], [22]. Such heterogeneous latencies often become a bottleneck in parallel applications.

Machine learning and graph problems are known to be highly parallel in nature, allowing great performance benefits to be acquired with parallel machines, such as Intel's Xeon Phi multicores [17]. However, large-scale multicores with NVRAM based main memory do not exist yet. There is a need to simulate such systems, and evaluate the heterogeneous latency challenge. Figure 1 shows the well-known parallel workloads, AlexNet [18] for image classification, and SSSP [11] for shortest path computations on the California road network. The simulations are run using an industry class simulator setup with a conventional DRAM, as well as an NVRAM with heterogeneous access latencies [24]. It can be seen that there is a significant degradation in performance due to excessive memory stalls in the the NVRAM setup.

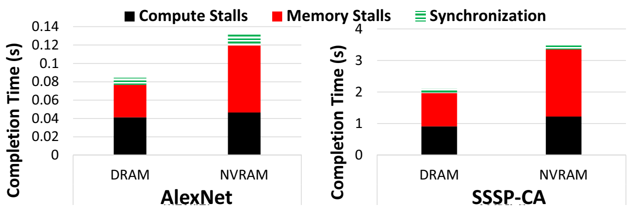

Figure 1: Completion time breakdowns with conventional DRAM and NVRAM based main memory.

Typical machine learning and graph applications rely on read-after-write patterns of temporally changing shared data [8]. In parallel implementations, threads see higher stall times due to the off-chip eviction of write data that is expected to be read afterwards. Moreover, the potentially large number of threads in futuristic multicore systems exacerbate the problem by increasing write contention on memory controllers. These data access patterns occur due to algorithmic aspects, such as layer-by-layer computations in machine learning and the iterative nature of graph algorithms. Since the problems operate on large amounts of data, the shared data structures get displaced by high reuse data. Therefore, high write latencies are observed by the NVRAM, and it becomes imperative to keep the shared data on-chip to mitigate computational costs.

Prior architectural works that attempt to keep data on-chip suffer from significant hardware complexity to distinguish data with high spatio-temporal locality from the write-intensive data. Reuse counters [9], cache line replacement [13], and cache partitioning [36], [38] schemes all attempt to keep data on-chip to minimize overall off-chip miss rates. However, they require fine-grain aspects such as counters at cache line granularity, thereby complicating hardware. Programmer-based software hints help reduce complexity by identifying application data structures that require resource allocations. Pragmas are applied within and across the target applications to forgo the need for complex control schemes.

This paper proposes a software–hardware framework that consists of profiling and exposing application data structures to the underlying microarchitecture. The hardware exposes last-level cache partitioning as a knob for data allocations. A set of static and dynamic learning paradigms are proposed that predict the allocation of cache partitions so as to co-optimize write and read miss rates in an NVRAM based multicore. Such performance prediction paradigms are known to help in various architectural paradigms [2].

Program profiling identifies structures in various benchmarks that potentially have variable read-after-write access patterns. Such structures are tagged as markers and pushed to the underlying cache hierarchy, where dynamically tagged cache partitioning is utilized at the hardware level to keep tagged data on-chip. A prediction model is proposed to cater for per-benchmark aspects that control the last-level cache partitioning. However, these aspects may change within different benchmarks across various phases of execution, and thus a fine-grain prediction model is also proposed that profiles application phases for resource allocations. It performs a design space exploration of tagged cache partitioning at the benchmark's function granularity. The learning is performed off-line using representative synthetic inputs for all benchmarks. The model is evaluated at runtime to predict the right cache partitioning strategy. The function-level partitioning model enables performance improvements at an average of 21% over an NVRAM system that does not utilize cache partitioning for managing the heterogeneous memory access latencies.

II. Related Work

The NVRAM latency problem has been known for some time, and various works have attempted to tackle it [12]. **Algorithmic works** rely on software changes to minimizes off-chip writes [5]. They introduce heuristics to stall writes for certain suspicious structures that may require off-chip data due to their increased data sharing. However, these works do not target applications that demonstrate unpredictably varying data reuse distances. Moreover, the strict implementations of various data analytic frameworks do not allow for algorithmic changes at each algorithm phase [1]. Various works also attempt to improve NVRAM access rates by lowering write latencies and improving bandwidth within the main memory. These include byte-addressing memory [22] and software management [29] for certain applications. These works do not learn for dynamic aspects, which leads to limited scope for performance improvements.

On-chip **architectural solutions** have been applied to improve data access latency. Cache partitioning is a well-known idea to reduce off-chip miss rates, thereby improving efficiency [34], [6]. These works target generic cache access rates, and are not entirely optimized for heterogeneous access latencies in NVRAM. Although some works have used cache partitioning to improve performance in NVRAM systems [37], [38]. However, such works do not evaluate large-scale multicores, where latencies are magnified by increased data accesses by concurrent threads. Other works improve cache replacement policies [38], [16], which require various hardware overheads to measure data reuse distances. Writeback-aware Cache Partitioning (WCP) [38] utilizes hardware based partitioning to improve access latency in NVRAM based systems. It utilizes LRU bits to predict write-backs from a cache slice for each cache line. Dynamic cache partitions are created to keep cache lines with higher amounts of write-backs, avoiding capacity issues that cause evictions to main memory. The constrained hardware resources do not fully exploit the temporally changing shared data, thus WCP acquires limited performance benefits. Moreover, the hardware requirements are complicated to implement, and processor vendors hesitate to implement them in production chips. What vendors do support is coarse-grain last level cache (LLC) partitioning, such as Intel's *Cache Allocation Technology* [15]. We propose to utilize existing resource allocation paradigms and build a software–hardware co-design framework. It utilizes cache partitioning as a control knob, along with software hints to *predict which data* needs to be kept on-chip to mitigate the challenges with NVRAM data access latencies.

III. NVRAM Challenges in Big-Data Applications

AlexNet is a winning application for ImageNet [10], an image recognition dataset, and is thus a viable contender for a case study of the proposed framework. It starts from a given input image, which propagates through its convolution filter layers and fully connected neuron layers to classify the image at the output. Neurons are distributed among threads for efficient concurrency control. Primary data sections for each layer contain input data, filter coefficients, and output data. Threads in a given layer read the propagated data obtained from the prior layer. However, threads have to read a lot of filter and image data to work on propagated output. In the case of AlexNet, the working set is $\sim$400MB, and unable to fit completely in the on-chip caches (up to 30MB in modern machines [15]). This requires data to be evicted to the main memory repeatedly. Moreover, in large-scale multicores, distributed cache slices are quite small for a given thread (tens to hundreds of KBs), and hence preserving locality is of utmost importance. This objective causes programmers to exploit locality on filter coefficients, and image data, as these are expected to exhibit high reuse. This locality is on *reads*, and given the constant access latency and bandwidth of DRAMs, such schemes provide ample performance. With less reuse on writes, data worked on in the previous iteration is evicted in favor of read data by conventional caching policies. Most of the latency seen through writes in a DRAM system is also hidden by the locality achieved via reads.

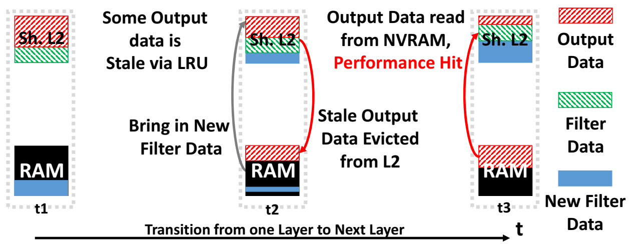
Sh. L2
RAM
t1
Some Output data is Stale via LRU
Bring in New Filter Data
Sh. L2
RAM
t2
Stale Output Data Evicted from L2
Output Data read from NVRAM, Performance Hit
Sh. L2
RAM
t3
Output Data
Filter Data
New Filter Data
Transition from one Layer to Next Layer
t

Figure 2: A read-write-read shared data access pattern in AlexNet, leading to expensive evictions to NVRAM.

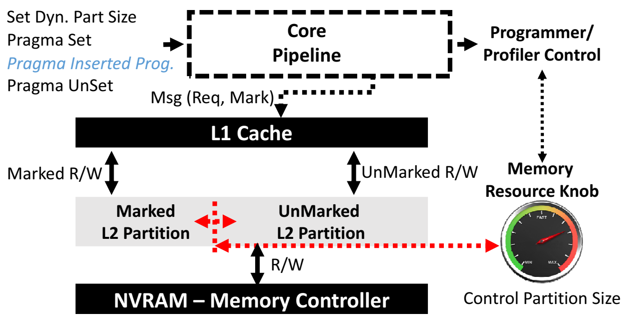

Figure 3: Data-marking based cache partitioning.

Algorithm 1 Various Layers of Parallel AlexNet

```
1: Init Matr. $input(in), filter/other(ch, s), outputs(out)$
2: //Red shows marked pragma set/unset around array accesses
3: for each Layer in range(0, nLayers) do
4:     Range = get_loc(tile)
5:     for each ch in range(0, kernel) do
6:         for each r in Range do
7:             Perform 2d conv. and accumulate
8:             $outC[r]$+=conv($out, image, in, ch, r, s$)
9:     Barrier
10: $outP[r]$ = PoolingLayer($outC, ch, s$)
11: Barrier
12: $outL[r]$ = LrnLayer($outP, ch, s$)
13: Barrier
14: for each neuron_layer do
15:     for each n in neuron do
16:         $outN[layer][n]$=neurMac ($outL, in, ch, s$)
17:     Barrier
18: Barrier
19: $outSoft[r]$ = SoftMaxLayer($outN, ch, s$)
```

However, in an NVRAM based architecture, the write latency is much higher (up to 10×) compared to the read latency in DRAM, and exploiting read locality on high reuse data can only help so much for performance. With smaller caches, writes build up on the memory controller queues, resulting in increased access latencies. Such cache evictions are caused by threads trying to exploit read locality by reading and prefetching read data from either the main memory, or the last-level cache slices of other cores. In the context of NVRAM, the lack of write locality causes high latency evictions and read stalls in various threads. This temporal aspect is depicted in Figure 2, which shows write data being evicted in favor of higher read locality data. Unpredictable temporal read-after-write patterns cause higher latencies to be seen on data that is supposed to be *read some time after being written to memory*. Moreover, real implementations that pipeline images into an AlexNet application observe extreme performance overheads since there is much more read locality to exploit.

This overhead can be mitigated by keeping temporal read-after-write data on-chip, so it can be accessed with lower latency. Software or even hardware prefetching is not expected to help much in such scenarios as it may cause pollution in shared caches, causing even further evictions. Off the shelf cache partitioning mechanisms can be utilized to develop a set of software profiling schemes. This ensures that only certain shared data with higher reuse distance gets captured without having to complicate hardware. It brings the programmer (or a profiler) into the loop to identify data structures and functions that require isolated last-level cache resources.

IV. Software–Hardware Framework

This section explains how data and functions are marked in software for cache resource allocation, as well as the architecture to facilitate the cache allocation scheme. Figure 3 shows the software and architectural aspects of the proposed scheme. Programmer marks data structures using *pragmas*. Set-partitioning is done to distinguish variable distance read-write data. The pragmas and set-partitions are propagated from the core to the cache hierarchy to partition the last-level cache. Several learning paradigms are evaluated to determine the right cache partition sizes for the marked and unmarked data. Software *pragma* insertions are done before profiling to allow optimal learning outcomes.

A. Software Marking in Applications

This section describes the process of marking data structures within a target application. Algorithm 1 shows the pseudo-code for AlexNet with various data structures and functions being utilized. The input matrix, in takes in images, and computed values from prior layers. The filter matrix takes in filter coefficients, which are large and change across layers. Finally, the output matrix stores computed values from the current layer, and is used in subsequent layers as the input layer. In machine learning algorithms, several data structures are accessed in various functions, with layers separated by synchronization barriers. It is easy to identify structures that are expected to be evicted due to long reuse distances. Such structures are separated by barriers or by locks, as parallel implementations require writes with consistency. In the case of AlexNet, these are all the *out*-based structures, and thus they are marked via program *pragmas* to be kept within the cache (Lines 8, 10, 12, 16, and 19 in Algorithm 1).

The *pragma* is set for all load and store instructions to marked data structures. In the proposed framework, *pragmas* are inserted for all output shared data structures. Barriers do not need marked resources as their variables are dynamically allocated to maintain high synchronization performance. The *Pragma* insertion is done by the programmer before any automated learning is performed. The compiler can also be modified to mark data structures at compile time, however that requires compiler analysis and is out of scope of this paper. Profiling and learning is then performed to determine the optimal partition sizes of marked and unmarked cache partitions.

B. Data Marking based Set Partitioning

Marking loads and stores is a primary solution to propagate software hints for data structures that are expected to exhibit temporal write locality during program execution. The cache allocation knob is tuned to allocate the right capacity for marked and unmarked data. Cache set partitioning is done at the last-level cache (LLC), where each LLC slice in a core gets partitioned based on sets. This is applied in a cache organization where private L1 caches are backed-up by shared L2 cache slices. The shared LLC is physically distributed among the tiles as L2 cache slices, and coherence is maintained using the directory based protocol. Set partitioning is selected due to its lower complexity compared to way partitioning [30]. For example, in an 8-way set associative $64KB$ L2 cache slice, there are 1024 total cache lines assuming a cache line size of 64 bytes. This results in 128 total sets, of which a percentage of sets are allocated to marked partitions. The remaining sets are used for unmarked partitions. Figure 3 shows the cache allocation knob, as well as the architectural aspects of LLC partitioning.

1) Pragma Propagation for Cache Partitioning: The *pragma* marked data structures are interpreted by the compiler as special load and store instructions. If the *pragma* is set and the corresponding data access misses the private L1 cache, it brings the respective cache line into the marked partition. Evictions from L1 to L2 cache slices are directed to the appropriate partition based on the tag lookup. The LRU policy is separately maintained for each partition in the L2 tag array, where one is for the marked sets and the other for unmarked sets [15]. If an L1 miss for marked partition results in L2 cache hit to unmarked partition (and vice versa), the L2 cache controller is responsible for moving that cache line to the appropriate partition. Moreover, partition sizes can also be dynamically changed during execution, which also results in moving cache lines between partitions.

2) Dynamic Management of Marked Partition: Cache partitioning is performed dynamically at per-function granularity to generate fine-grain control of the cache resource allocation knob. This is also propagated to the architecture via an added

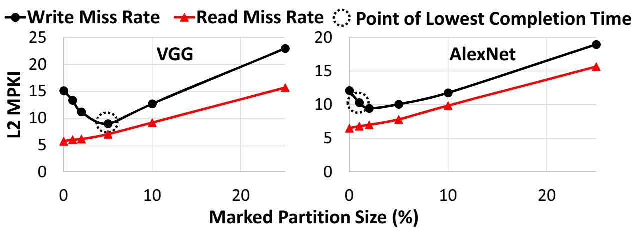

Figure 4: Read and write miss rates for various marked partition sizes. Optimal points are observed at different marked partition sizes.

instruction directed by another program-level *pragma*. The L2 cache slice access is stalled and sets are partitioned according to the propagated value. This is utilized as a global parameter, and L2 slices of all cores are uniformly partitioned with equal sized partitions. Sliding the partition scale in Figure 3 means that unmarked cache lines may start polluting the marked partition if the marked partition size is increased, and vice versa. Prior architectural works solve this issue by either flushing the polluted sets to main memory, or moving cache lines when a cache partition size changes. Flushing adds further overheads for NVRAM as it flushes cache lines to the memory controller queues, thereby increasing write traffic. Moving cache lines is a more viable solution, and can be executed only when a cache line is brought into a partition that has been polluted. It is further optimized to only move a cache line upon an L1 miss. From here the L2 controller receives a load or store for a marked partition, and sees if it is already in the marked partition. If the cache line is incorrectly in the unmarked partition, the L2 controller moves the cache line to the marked partition. Cache line movement requires stalling of the L2 cache slice for atomicity and consistency purposes. Thus, upon a function change within an application, the learner sets a cache partition size value and the cache stalls to move cache lines on L1 misses.

C. The Effect of Partitioning Knob on LLC Misses

Due to the heterogeneous access latencies associated with NVRAM, the goal is to minimize *write miss rates* while keeping read misses in check. Figure 4 exhibits the L2 read and write misses per kilo instruction (MPKI) associated with various marked partition sizes, for a 256 core multicore with a 64KB L2 cache slice per core. The MPKI versus partition size results are shown for two machine learning workloads, AlexNet and VGG [1]. VGG is a machine learning workload with a larger memory footprint than AlexNet, and hence shows varied sensitivity to resource allocations. The write MPKI decreases due to reduction of evictions acquired by allocating cache memory to marked partitions. However, the read MPKI goes up with the increase in marked partition size as more capacity is taken out of the already exploitable read locality. Overall, misses for both reads and writes increase as larger marked partitions decrease capacity of unmarked

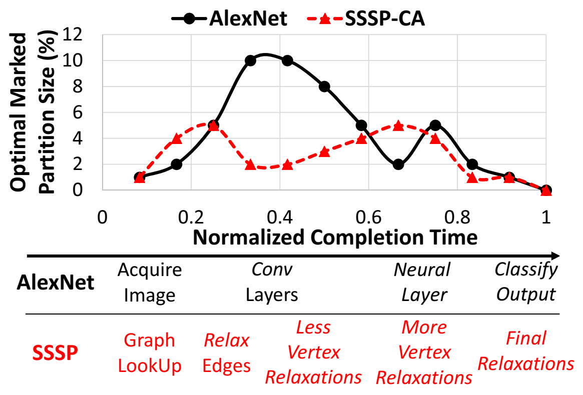

Figure 5: Completion times of various benchmark phases normalized to their maximum completion time. Data shows the need to dynamically control marked partition sizes based on application phases.

partitions.

VGG optimizes at a higher marked partition size than AlexNet, primarily because of more high distance reuse data in VGG due to it being a larger and wider network with increased memory requirements. Optimal performance is observed at a point where write miss rates are not minimized, which happens because of the increase in read misses that need to be traded-off. This favors a paradigm that can configure these architectural knobs *dynamically* at *per benchmark* granularity. However, even per-benchmark partition settings may not be optimal as different *phases within a benchmark* exhibit varying memory requirements. This dynamic resource allocation requirement must be adopted at runtime. Hence, several static and dynamic schemes are presented and evaluated in this paper.

D. Static Partitioning Allocation

To maximize performance, the programmer can assign static partitioning size across all benchmarks. This is achieved via program profiling heuristics to acquire an optimal partition size from all target benchmarks. All benchmarks are run for various partition sizes, and the partition size that gives the best overall performance is selected. This approach is expected to under-perform, as a single partition size might not work for all benchmarks. This is evident from Figure 4, where optimal partition sizes change across the two benchmarks. A per-benchmark or a per-phase learner is desirable for near-optimal performance.

E. Per-Benchmark & Per-Phase Partitioning Allocation

The proposed per-benchmark learner optimizes partition sizes individually for each benchmark. For example, for VGG in Figure 4 the optimal partition size is 5%, while for AlexNet it is 1%. This helps for a benchmark with characteristics common to all its inputs. However, variable benchmark aspects must be taken into account. These aspects

Machine Learning		**Graph Workloads**	
convolution	get_loc	reduction	relax
neural_mac	EcludianDistance	find_component	graph_lookup
nonlinear_conv	SoftMaxLayer	reconstruct	find_nearest_neighbor
tilingN	pooling	add_to_tri_count	search_vertex
LrnLayer	PushForward	calc_dangling_PageRank	find_community

Figure 6: Functions used for the per-phase learner.

are considered inputs to the model, while resource allocations are considered outputs of the prediction model. The model is trained on synthetic inputs, and then benchmarks are scheduled online for the partition size selection.

While per-benchmark partitioning is expected to reduce miss rates, workloads have various diverse functions, each of which have sliding memory requirements. These traits are expected to vary partition knobs throughout program execution. Figure 5 shows these variations for AlexNet and SSSP running California road network, with the marked partitioning size manually changed to optimal at function call granularity. In AlexNet, these calls constitute functions like *get_loc ()* and *conv ()*, as depicted in Algorithm 1, and many others not shown in the pseudocode. In SSSP, these constitute graph *lookups* and *relax* routines. It can be seen that the optimal partition size per function varies during execution from 0% to 10%. Figure 5 also shows the functions that dictate a particular optimal partition size. AlexNet has higher marked partition size requirement for *convolution*, and the *neural* network fully connected layers. SSSP, however has variations in partition size requirements due to its iterative processing of the input graph (roadNet-CA). Early in execution, SSSP *relaxes* on several vertices, and hence needs more marked cache resources due to variable reuse distances of many vertices. These updates sublime and then peak again in Figure 5 when more vertices are activated for relaxations, depending on the connectivity of the input graph. Such variations add dynamic aspects that need to be learned at a per-function granularity. This learner paradigm is formulated as follows.

Model Variables are `Benchmark` and `Input` combinations, and `Partition Sizes`. The output constitutes an optimal marked partition size for a given function, with the learner optimizing for performance. Benchmarks are formulated into two vector variables, $\vec{B}1$ and $\vec{B}2$, which distinguish various machine learning and graph workload functions. As the model learns on several benchmarks and functions/kernels within benchmarks, it is imperative to reduce learning complexity. This is done by marking and learning during the offline profiling phase of the proposed framework. For example, graph lookups are common across all graph workloads, as are reductions and other parallel functions. In machine learning workloads, these are getting kernel filter ranges (*get_loc()*), *conv()*, and *neuMac()*). Bench-

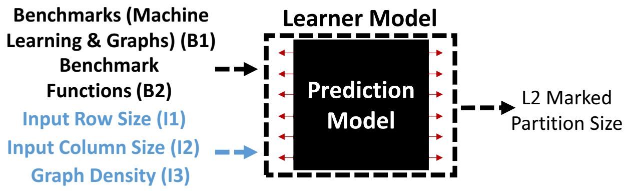

Figure 7: Learner model encompassing inputs and outputs.

mark variables are thus integers where for example for $\vec{B}2$, different functions have different values depending on their importance in the program. Figure 6 shows the different functions for the evaluated benchmarks.

Benchmark inputs also vary performance, and thus need to be considered when determining partition sizes. For example, larger inputs may not fit in naively sized partitions, while smaller inputs may under-utilize cache resources. Inputs to all target benchmarks take the form of matrices, which can be classified via row and column sizes, and these are taken as two input vector variables ($\vec{I}1$ and $\vec{I}2$). The $\vec{I}3$ input for density (edges per vertex) is considered as an additional input variable for graph algorithms.

Learning Paradigms: Multiple non-linear and linear regression models, as well as several deep learning models are constructed from the above $\vec{B}$ and $\vec{I}$ variables. Due to non-linear aspects depicted in Figure 4, a non-linear higher order regression model is required to achieve suitable learning capability. A linear regression is also evaluated for comparison, although it is not expected to show significant performance improvements.

Multi-layer deep learning perceptrons (MLPs) are also utilized for learning, as they are known to learn well on non-linear aspects. The number of neurons and layers is dictated by the required classification accuracy, and the number of inputs for classification [35]. Hence, MLPs with at least 32 neurons (MLP-32) (4 layers, 8 neurons per layer) are used in our case. The inputs and outputs of all evaluated learners are the same as shown in Figure 7. The proposed models and learners are also compared with an ideal learner that exhaustively evaluates the search space for optimal cache partition sizes.

Training Model: The learner models are trained using synthetic inputs to the benchmarks. Both regression and deep learning models are trained with the same inputs as they have the same input and output variables. Synthetic inputs to machine learning workloads are matrices and arrays (i.e., various kernels and input/output layers) that vary in size, while graph benchmarks are trained for uniform random graphs [4], and Kronecker graphs modeling power law mechanics of real graphs [20]. These inputs are varied from a size of 1 to 4096 for variable-dimensional matrix sizes, and 1 to 1M for array sizes. Synthetic graph inputs are varied from 16 to 16M vertices, and 1 to 1024 edges per vertex. These sizes are well within the sizes observed in real world input graphs.

Large complexities result from these tuple variables, as many combinations can be made from the input variables and output partition sizes for various benchmarks. With 13 benchmarks, more than 50 profiled functions, several hundred synthetic training inputs, and several partition sizes to choose from, the total complexity results in several million combinations. Due to this large complexity, training is done offline. After training, the predictor is deployed to process real inputs. The proposed predictor is deployed in software to take in variables to predict the marked partition sizes.

The Importance of Model Variables: How the model trains for the target variables can be seen in the equation 1, which shows a 6th order non-linear function that conforms to more than 90% classification accuracy.

$$F(Cache\ partition\ size) = w1(\vec{B}1)^6 + w2(\vec{B}2)^3 + w3(\vec{I}1)^6 + w4(\vec{I}2)^4 + w5(\vec{I}3)^2 + 8.0 \tag{1}$$

Variables with more direct correlation to performance, such as benchmark type ($\vec{B}1$), and input row size ($\vec{I}1$) have higher orders. As seen earlier in Figure 5, phase behavior does have a role in performance. Phase changes within benchmarks are shown by $\vec{B}2$, where weight changes symbolize function changes, which changes intensity of partition sizes required for optimality. Changes in phases correspond to changes in weights, which changes output partition sizes in the regression equation.

Graph workloads have more randomly skewed memory access compared to machine learning workloads. From input variables, $\vec{I}1$ has the highest order, as input row sizes (matrix row counts in machine learning, and vertex counts in graphs) directly affect required concurrency and memory requirements. $\vec{I}1$ also defines cache requirements, which correlates directly with L2 partition size requirements. $\vec{I}2$ and $\vec{I}3$ also affect partition size requirements in indirect ways and hence have some weight on the output partition size. Similarly, a multi-layer perceptron learner is expected to have similar trends in terms of neurons and layers. The overhead of these learners is added to the overall completion time during the evaluation phase. A sensitivity study is done in the evaluation section to quantify which learner provides the most performance benefit at acceptable classification accuracy.

V. Evaluation Methodology

A. Simulator Setup

We evaluate a futuristic 256–core tiled multicore processor with a two-level private–shared cache hierarchy. As hundreds of cores do not exist in real machine setups with NVRAM based main memory, we utilize a simulator setup to get

Architectural Parameter	Value
Cores	256 RISC-V @ 1 GHz
Comp. Pipeline per Core	In–Order, Single–Issue
Word Size	64 bits
Memory Subsystem	
L1–I Cache per core	16KB, 4–way Assoc., 1 cycle
L1–D Cache per core	16KB, 4–way Assoc., 1 cycle
L2 Inclu. Cache per core	64KB, 8–way Assoc. 2 cycle tag, 4 cycle data
Cache Line Size	64 bytes
Directory Protocol	Invalid. MESI, ACKwise$_4$ [19]
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention, 64-bit Flits (Infinite input buffers)
Flit Width	64 bits
Memory Controllers - NVRAM Parameters	
Num. of Mem. Contr	8 - Dual Channel
Mem. Contr. Buffer Size	64 entries per Channel
Bandwidth	20 GBps per Controller
NVRAM Read-Write Lat.	100ns-1us
DRAM Latency	100ns

Table I: Architectural parameters for evaluation.

the best possible understanding of the NVRAM latency challenge. The proposed system is implemented in an in–house industry–class simulator which uses open source RISC-V cores and associated LLVM compiler intrinsics. The default architectural parameters used for evaluation are shown in Table I. The total on-chip cache size amounts to $24MB$ of which $16MB$ is dedicated to the shared last-level cache that is distributed among the per tile L2 cache slices.

NVRAM parameters have been acquired from prior works and real systems [25], which consist of memory controller parameters for channels and controller buffers, as well as the heterogeneous memory access latency and bandwidth. Prior works show that the disparity between read and write NVM latencies (for Phase Change Memories (PCMs) or Spin based memories (STTRAMs)) ranges around 10×, and we thus keep this disparity in our setup as well [27]. An $8GB$ of NVRAM memory capacity is modeled. The simulated system utilizes either NVRAM or DRAM, but not both at the same time. However, a study is also done to show a DRAM-only analysis.

The on-chip network is modeled with 2–cycle per hop delay and XY routing protocol. The appropriate pipeline latencies associated with loading and unloading a packet onto the network are accounted. In addition to the fixed per–hop latency, network contention delays are also modeled, which are derived from Graphite [26]. The energy numbers are obtained from McPat [23] using $22nm$ technology scaled to $11nm$. Read and write energy numbers from [7] are utilized for each NVRAM access.

Benchmark	Input Dataset
Machine Learning	
MLP, CNN-MNIST, KNN [28]	MNIST [1]
CNN-GTRSB [31]	GTSRB [33]
AlexNet [18], SqueezeNet [14], VGG[1]	ImageNet [1]
Graph Analytic CRONO Suite [3]	
PageRank, Triangle Counting, Community, Connected Components, SSSP, BFS	California Road Network [21]

Table II: Benchmarks and inputs.

B. Benchmarks and Evaluation Metrics

The benchmarks and their inputs are presented in Table II. Seven machine learning benchmarks are developed using the models and datasets referenced in Table II. These consist of MNIST, which is used for handwritten digits identification; GTRSB, which is used for traffic signs for detection; and ImageNet that has images to be classified. In terms of workloads, we incorporate K-Nearest Neighbors (KNN) and an MLP based neural network, which has 768 neurons and multiple layers. CNN adds convolution layers to MLP based networks to improve classifications. AlexNet and VGG are industry class workloads for image classification, and comprise of convolution layers and neural networks, with VGG being larger than AlexNet. SqueezeNet is an inception neural network with a reduced model size compared to AlexNet, rendering it suitable for architectures with smaller caches. Six graph benchmarks are also taken from the parallel *CRONO* benchmark suite [3].

Each benchmark is run to completion, and the completion time and dynamic energy consumption of the *parallel* region are measured. With the learner models parallelized, they take between $1ms$ to $10ms$ for the evaluation phase. The learner's runtime evaluation overheads are added to the overall completion time for fair comparisons. For the per-benchmark scheme, the learner overhead is added once per workload run, while for the per-phase scheme it is added on function calls. The static scheme does not have any dynamic learner overhead. These schemes and learners are also compared to an ideal case which does not have any performance overheads and has 100% classification accuracy.

VI. Evaluation

This section analyzes the proposed framework in terms of performance and energy improvements. All results are normalized to a system with main memory (DRAM or NVRAM) without any LLC partitioning scheme. The per-function learner utilizes MLP-64, while the per-benchmark learner utilizes the MLP-32 deep neural network. These learners are selected as they give the highest performance for the two settings. A hardware-only scheme, WCP [38], is also compared in terms of performance and energy. WCP utilizes LRU aspects to predict cache lines with high probability of writebacks, so they can be moved to marked partitions to reduce evictions (scheme explained in Section II).

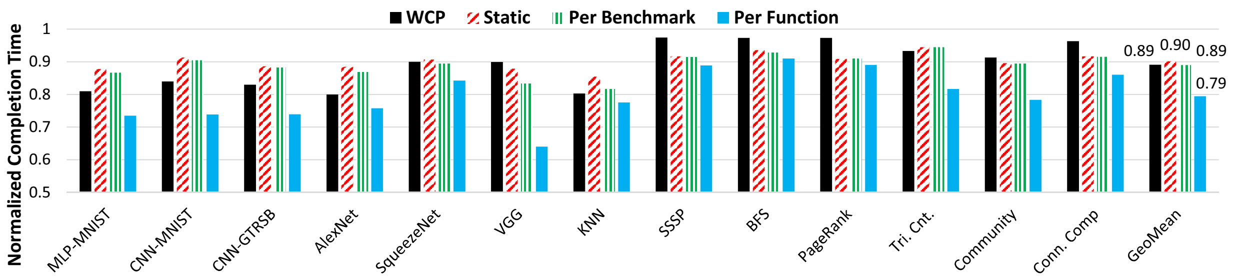

Figure 8: Normalized completion times with the proposed system over a scheme that does not partition the LLC.

A. Performance Improvements

Two cache partitioning schemes (static and dynamic) are discussed in Section IV, where dynamic includes per-benchmark and per-function granularity based LLC partitioning. Figure 8 shows the results for the proposed NVRAM based system. On average, the acquired performance improvement using the static scheme is ∼10%, while the per benchmark and per function schemes improve performance by 11% and 21% relative to an NVRAM system that does not partition the LLC. WCP performs well for machine learning workloads due to distinguishable structures that have higher write-backs. However, it does not perform as well for graph workloads, where cache lines are packed with vertices with dynamically varying access requirements. WCP is also not optimized for *large-scale multicores*, as such parallel machines put more pressure on the limited memory controllers and the LLC due to more requests from larger numbers of threads. This enables low distance write locality to be exploited by WCP, whereas the problem in machine learning and graph workloads pertained to high distance writes, and hence WCP does not perform as well.

Compared to the smaller network SqueezeNet, larger neural network workloads, such as VGG provide larger performance gains. This is because larger working sets do not fit in small shared caches, which exacerbates the capacity and high distance reuse eviction problem. In the case of KNN, there is a lot of data with high reuse distance, and hence this results in larger benefits. Graph workloads also give improvements, with all workloads with at least 12% benefit when using the per function scheme. Workloads that have smaller working sets due to smaller dynamic data structures (such as BFS) do not see large gains as their working set already fits in shared caches. Moreover, workloads with phases, such as reductions in Triangle Counting also see improved gains from the dynamic scheme as they temporally change their memory requirements. However, the total benefit acquired in graph workloads is much lower compared to that achieved in machine learning. This happens primarily because graph workloads incorporate unstructured access patterns, and hence they cannot exploit spatial locality in marked partitions.

Overall, the static and the per-benchmark schemes do provide performance benefits. However, more benefits can be acquired in workloads that require different resource allocations across phases. All analyzed schemes are within 5% of the ideal case, which shows the effectiveness of the learner. One downside of the per-function scheme is that programmers have to mark $functions$, however this depends on how much performance is required to be traded-off with marking complexity.

B. Energy Improvements

The rise in read accesses due to the proposed scheme may induce additional energy consumption, and thus it is important to analyze dynamic energy of the system. Figure 9 shows the energy benefits, where trends follow performance results. Energy reductions stem from less writes going off-chip to NVRAM, which has a higher per-write energy cost, and also from the reduction in completion time. In machine learning, larger networks such as VGG improve more on energy than smaller networks such as SqueezeNet because smaller networks do not suffer much from the lack of cache capacity. As more energy is consumed in contended functions, such as reductions, workloads with these functions acquire more energy benefits. For graph workloads, phases matter as they have reductions that consume a lot of energy via accesses, which can be improved using optimal cache allocation. On average, a 51% energy benefit is acquired with the per-function scheme over an NVRAM system with no partitioning, while the static and per-benchmark schemes provide 25-29% benefits. All proposed schemes are again within 5% of the ideal case. WCP only provides a 23% energy benefit, which is lower than the benefit acquired by the static scheme. This is because WCP does not perform well in graph workloads, where access patterns are input dependent and require intricate reuse distance information.

C. Learning Models

The target learning model for cache resource allocations is predictive, and thus it must be compared with ideal results to evaluate its effectiveness. Table III shows various learning schemes to predict partition sizes at phase and benchmark granularities. The proposed model is compared with regression schemes (R-order), other multi-layer perceptrons (MLP-neurons), and an ideal learner with no overhead and ideal accuracy. Table III also shows the average overhead of each evaluated scheme, which increases with more complex learners and with the function level scheme.

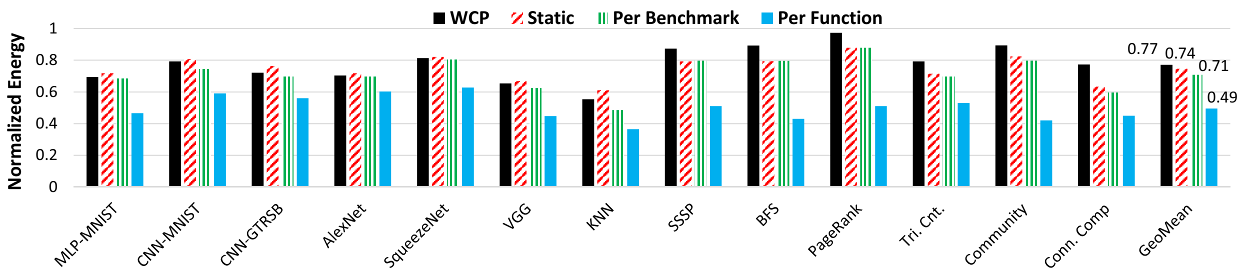

Figure 9: Normalized energy acquired using the proposed system over a scheme that does not partition the LLC.

Table III: SpeedUp (Speed.), accuracy (Acc.) shown for each learner. The overhead (Over.) is the computation overhead at each invocation of the learner. Settings in bold show the deep-learners used for the per-benchmark and the per-phase models.

Setting	Per Bench.			Per Phase.		
	Speed.	Acc.	Over.	Speed.	Acc.	Over.
	%	%	(ms)	%	%	(ms)
Lin.(R-1)	2.2	48.4	0.5	4.9	44.8	12.4
R-4	4.1	65.3	3.0	7.3	62.5	24.4
R-5	8.1	84.3	3.4	12.3	82.5	27.7
R-6	9.8	82.1	3.8	14.1	84.5	32.1
MLP-32	**11.4**	**91.3**	1.8	17.6	86.1	15.3
MLP-64	10.8	93.5	2.1	**20.8**	**90.3**	17.0
MLP-128	9.5	95.3	2.9	20.4	92.6	24.1
Ideal	12.2	100	0	22.1	100	0

The proposed per-benchmark and per-phase schemes provide ample performance benefit with their target neuron settings. It is also within 5-10% of the ideal learner, meaning that it can be deployed for such benchmarks. Linear regression does not provide ample benefits, primarily due to the non-linear aspects of input dependence and partitioning settings. Higher order regressions perform well, but exhibit higher overheads. MLP-128 does not perform as well as MLP-64 or MLP-32 because it incurs a huge evaluation overhead due to higher neuron complexity. While the per-phase scheme shows a higher overhead due to more function and learner calls, it does provide enough benefits to offset these overheads.

D. Memory Controller Queuing Delays

The proposed hardware-software scheme is deduced to reduce write misses, which in return should reduce contention on memory controller queues. Figure 10 shows this percentage improvement in queuing delay over the system utilizing no partitioning. It is seen that machine learning workloads that have higher write misses due to variable distance write data, show the most improvement in contention reduction. This result is in line with the performance and energy improvements acquired in Figures 8 and 9.

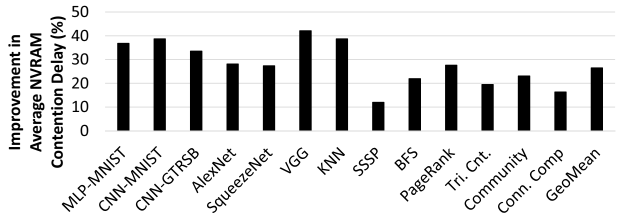

Figure 10: Improvement (%) in average NVRAM queue contention delay over the baseline utilizing no partitioning.

Table IV: Write latency variations for various DRAM and NVRAM systems.

Main Mem Type	DRAM	NVRAM			
Write Latency (us)	0.1	0.2	0.5	0.7	**1.0**
GeoMean SpeedUp (%)	6.0	11	15	18	**21**

E. Write Latency Sensitivity for NVRAMs

With upcoming NVRAM technologies from various industry leaders, it is imperative to analyze various write latencies associated with our proposed scheme. Different write latencies are thus compared, ranging from 2× to 10× worse compared to the read latency. Table IV shows the performance results across all benchmarks acquired with various write latencies. It can be seen that the acquired performance benefits variy from 6% with the DRAM-based system to 21% in an NVRAM-based system that has a 10× worse write latency than reads. Intermediately worse write latencies also show performance improvements that are directly correlated with the latency penalty. This shows that the proposed scheme works effectively for various DRAM and NVRAM based systems executing machine learning and graph analytic parallel applications.

VII. Conclusion

NVRAM based main memory architectures have high write latencies that induce performance and energy overheads in machine learning and graph workloads. This work improves performance in such systems by intelligently marking data structures in software that have temporally variable reuse distances during program execution. Dynamically changing last-level cache partitions are created to keep marked data on-chip, thereby reducing off-chip evictions of write accesses. This work shows that performance benefits of 21%, and energy benefits of 51% can be acquired using the proposed dynamic learning paradigm for LLC partitioning in futuristic multicores.

VIII. Acknowledgments

The authors thank Dong Chen (IBM Research), who helped motivate and enable this work. This research was partially supported by the National Science Foundation under Grant CNS-1718481. This work was also supported in part by the US Government under a grant by the Naval Research Laboratory.

References

[1] R. Adolf, S. Rama, B. Reagen, G. y. Wei, and D. Brooks. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

[2] M. Ahmad, C. J. Michael, and O. Khan. Efficient situational scheduling of graph workloads on single-chip multicores and gpus. *IEEE Micro*, 37(1):30–40, Jan 2017.

[3] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE Int. Symp. on*, pages 44–55. IEEE, 2015.

[4] David A. Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite, 2006.

[5] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri. Write-avoiding algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658, May 2016.

[6] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412, New York, NY, USA, 2014. ACM.

[7] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. January 2011.

[8] Y. Chen and et. al. DaDianNao: A machine-learning supercomputer. In *47th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 609–622, 2014.

[9] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[11] H. Dogan, F. Hijaz, and et. al. Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging. In *2017 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 254–264, May 2017.

[12] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale. On the role of nvram in data-intensive architectures: An evaluation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 703–714, May 2012.

[13] Fei Guo and Yan Solihin. An analytical model for cache replacement policy performance. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 228–239, New York, NY, USA, 2006. ACM.

[14] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[15] Intel. Intel® 64 and ia-32 architectures software developer's manual volume 3b system programming guide, part 2. 2015.

[16] Aamer Jaleel and et. al. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual Int. Symp. on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[17] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang. Training large scale deep neural networks on the intel xeon phi many-core coprocessor. In *2014 IEEE Int. Parallel Distributed Processing Symp. Workshops*, pages 1622–1630, May 2014.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Adv. in Neural Inf. Proc. systems*, pages 1097–1105, 2012.

[19] George Kurian and et. al. Atac: a 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 477–488. ACM, 2010.

[20] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[21] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[22] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 945–956, May 2012.

[23] Sheng Li and et. al. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd Annual IEEE/ACM Int. Symp. on Microarch.* ACM, 2009.

[24] Micron. NVDIMM: Persistent Memory Performance. https://www.micron.com/~/media/documents/products/product-flyer/nvdimm_flyer.pdf, 2016.

[25] Micron. Breakthrough nonvolatile memory technology, 3dx-point. 2017.

[26] Jason E Miller and et. al. Graphite: A distributed parallel simulator for multicores. In *High Performance Comp. Arch. (HPCA), IEEE 16th Int. Symp. on*, pages 1–12. IEEE, 2010.

[27] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, May 2016.

[28] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[29] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the nvram era. *Proc. VLDB Endow.*, 7(2):121–132, October 2013.

[30] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proc. of the 38th Annual Int. Symposium on Computer Architecture*, 2011.

[31] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 Int. Joint Conf. on*, pages 2809–2813. IEEE, 2011.

[32] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proc. of the 44th Annual Int. Symp. on Computer Architecture*, ISCA '17, pages 175–186, NY, USA, 2017. ACM.

[33] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *Neural Net. (IJCNN), The 2011 Int. Joint Conf. on*, pages 1453–1460. IEEE, 2011.

[34] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[35] Yoshiyasu Takefuji. *Neural Network Parallel Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[36] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 652–665, New York, NY, USA, 2017. ACM.

[37] Wei Wei, Dejun Jiang, Jin Xiong, and Mingyu Chen. Hap: Hybrid-memory-aware partition in shared last-level cache. *ACM Trans. Archit. Code Optim.*, 14(3):24:1–24:25, 2017.

[38] M. Zhou, Y. Du, B. R. Childers, R. Melhem, and D. Mossé. Writeback-aware bandwidth partitioning for multi-core systems with pcm. In *Proc. of the 22nd Int. Conf. on Parallel Arch. and Compilation Techniques*, pages 113–122, 2013.