Tree-based read-only data-chunks for NVRAM programming

Kumud Bhandari
Department of Computer Science
Rice Unviversity
Houston, Texas 77005
Email: kumudb@rice.edu

Vivek Sarkar
Department of Computer Science
Rice Unviversity
Houston, Texas 77005
Email: vsarkar@rice.edu

***Abstract*—As the DRAM technology is fast approaching its scaling threshold, emerging non-volatile, byte-addressable memory (NVRAM) is expected to supplement and eventually replace DRAM. Future computing systems are anticipated to have a large amount of NVRAM, possibly spanning across more than one coherence domain. Furthermore, taking advantage of in-place persistence provided by the NVRAM in future systems requires a strategy to prevent tolerated failures (e.g. power failure) from leaving persistent data in an incoherent state. A fresh look at memory management approaches across the system stack is required to fully utilize future NVRAM. In this paper, we carefully assess the NVRAM-related memory access and management challenges, its implication to application level programming, and examine the suitability of tree-based read-only data chunks to NVRAM programming.**

I. Introduction

For the past few years, current DRAM technology scaling has been in jeopardy as the DRAM chip manufacturers have struggled to scale beyond 40nm [1]. Consequently, researchers have been pursuing various alternative memory technologies to overcome the current limitations. Incidentally, almost all

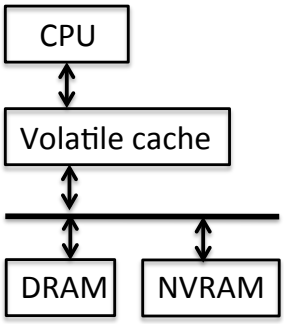


Fig. 1: Future NVRAM system

emerging next generation memory technologies such as Phase Change Memory(PCM) [2], Memristor [3], and 3D XPoint [4] are non-volatile in nature. These NVRAM technologies combine the byte-addressability of a DRAM with the persistence traditionally provided by the hard drive. Byte-addressable NVRAM enables persistent data to be accessed using CPU load and store instructions and the persistent data to be stored in the same format as it is manipulated. This is in contrast with the filesystem-backed persistence where a certain amount of de-/serialization overhead has to be incurred. The access latency of NVRAM is expected to be eventually comparable to DRAM, and the use of volatile cache hierarchies will continue as usual to reduce the latency of accessing transient and persistent data stored in DRAM and NVRAM respectively. Fig.1 shows the memory hierarchy of future systems with NVRAM.

The current system stack is less than suitable for fully exploiting the next generation non-volatile memory. Experts anticipate that NVRAM will scale past the current 256 TiB limitation imposed by 48-bit virtual addressing in foreseeable future. Interestingly, future systems such as HP's "The Machine" [5] is expected to have non-volatile memory spread across multiple coherence domains. In the absence of proper programming model, programmers will be burdened with reasoning about data coherence in parallel applications running on these systems. Additionally, in order to take advantage of direct persistence provided by NVRAM programmers have to concern themselves with the safety of the data in NVRAM when a failure interrupts a set of updates being applied to persistent data. Partial updates being cached in volatile caches (and thus unable to survive a failure) may introduce inconsistencies to persistent data such as dangling pointer etc [6]–[8].

Various filesystems and databases have dealt with failure-induced inconsistencies in persistent data using transactional and copy-on-write approach. Our work differs from these lines of work as our work focuses on making persistence a first-class citizen within a programming language. Prior work in orthogonal persistence [9] which pursued this goal predates NVRAM and hence suffered from technology impedance. Furthermore, in the context of orthogonal persistence, all data is allocated in DRAM, and the persistence of data is based on reachability analysis at certain program points, leading to a large space and performance overheads. Unlike orthogonal persistence, our work focuses on persistence designation of data through explicit allocations. The allocated persistent data is then updated using fail-safe mechanism. When the application restarts across system recycle or failure, small user-written restart code reinitializes the program context from persistent data and the computation resumes. Prior works, such as Atlas [6] and Mnemosyne [7], have pursued this persist-and-reuse

model in the context of NVRAM. They offer C/C++ NVRAM programming extensions which enable a programmer to explicitly allocate persistent objects and update them in a fail-safe manner. However, it requires a programmer's familiarity with the new programming constructs and paradigms, has large runtime failure consistency overheads, while still requiring programmers to reason about a coherent update to persistent data possibly spread across multiple coherence domains in NVRAM.

Our goal in this paper is to demonstrate that the Freshbreeze-inspired [10] ***tree-based memory (TBM)*** architecture (with read-only data blocks) along with a suitable caching policy and an automatic memory management scheme can support persistence as a first-class citizen within a familiar ***task-structured parallel programming languages (TSPL)***, such as HJ, with no additional programming burden (other than the explicit allocation of persistent memory), or runtime overheads. We discuss in details various programming challenges associated with NVRAM and how TBM architecture addresses these challenges. Our paper makes the following contribution:

- careful assessment of the programming challenges associated with NVRAM systems,
- preliminary feasibility study of a tree-based memory architecture in the context of NVRAM,
- an appropriate caching policy and a scheme for automatic management of persistent and transient memory in the context of a TBM architecture, and
- a library-based emulator for further study of a TBM architecture.

II. NVRAM PROGRAMMING CHALLENGES

A. Failure consistency overhead and complexity

Given volatile caches, and CPU reordering of memory writes, updates to persistent data may not appear in correct order in NVRAM. To ensure the consistency of persistent data, a certain set of updates to persistent data need to be applied in all-or-nothing basis w.r.t. failure. Consider a simple program in fig. 2. In this example, some persistent memory is being allocated, initialized, and published (made reachable from some persistent roots) in statements 1, 2 and 3 respectively. Updates only from statements 1 and 3 becoming visible in NVRAM due to a failure will leave persistent data in an inconsistent state as the persistent pointer, p_root will refer to an uninitialized persistent memory location upon a restart.

```
: int t[]=nvm_new int[5];
: initialize (t);
: p_root = t;
```

Fig. 2: Sample NVRAM program

Much of the previous work has focused on providing a mechanism for programmers to specify a set of updates to persistent data having all-or-nothing effect w.r.t to failure. Almost all available NVRAM programming libraries rely on a log-based approach to enforce failure atomicity [6]–[8], [11]. Logs are written and made visible in NVRAM before updates are applied to persistent data. Hence, persistence comes at the cost of having to write logs and flush them out of volatile caches[1]. Additionally, all new modifications to persistent data which belong to the failure-atomic set of updates must also be flushed out of the cache before a failure-atomic set of updates is deemed complete. Results presented in [6] show that persistence is obtained at around 20-30% performance overhead on average compared to transient applications.

B. Persistent data across multiple coherence domain

Given the projected size of the available memory in future systems, it is likely that data will be spread across more than one coherence domain [5] due to the scaling limitations of coherence protocols [12]. Previous works in NVRAM programming (see § II-A) assume that the persistent data is stored within the same coherence domain. It is highly likely that programmers may have to accommodate for the inconsistencies arising from updates to persistent data across multiple coherence domains in future systems.

C. Persistent memory addressing

In NVRAM, data is stored in the same format as it is manipulated. As persistent data survives beyond the lifetime of the process, pointers within persistent data have to be valid as well. No good solution has been proposed so far to handle persistent pointers. Almost all previous work proposes persistent data to be organized in a named-container similar to a mmaped file, often referred to as *NVRAM regions* [6]. Numerous previous works [6], [7] assume that such NVRAM regions are always mapped to the same virtual address, thus enabling absolute pointers stored inside an NVRAM region to be always valid [6]. Such approach requires a system-wide enforcement that no two NVRAM regions (even though created by separate processes) ever have an overlapping virtual address region. It is unclear how this can be realistically enforced in a large scale systems. Some other work advocates using a relative addressing within an NVRAM region [11]. This approach wastes expensive CPU cycles to calculate an effective absolute address for each persistent memory access [11].

D. Dynamic management of persistent memory

Consider persistent updates from only statement 1 in fig. 2 being visible in NVRAM due to a failure. Without the proper memory management framework, this would result in a permanent memory leak. Such permanent memory leaks in NVRAM is more disastrous than in DRAM because it can accumulate over several execution cycles. In much of the previous work, the burden is on programmers to avoid permanent memory leaks. NVRAM programming libraries, such as Atlas, require programmers to ensure that all memory allocations occurs only

[1] a single cache line flush is $\sim 200ns$ in Intel x86 arch irrespective of whether the line is dirty or clean

within a failure-atomic section to avoid memory leaks. Memory leaks due to programming errors are unmitigated in almost all published NVRAM programming libraries with the exception of NV-Heap. So far, none of the NVRAM programming libraries offers a robust solution against permanent memory leaks.

III. Tree-based memory

In this section, we describe the tree-based memory (TBM) hierarchy and explain how such a memory architecture addresses NVRAM programming challenges discussed in § II. Prior works in the recent past such as HICAMP [13] and Fresh Breeze [10] have explored memory organized as a tree of related read-only data chunks. However, previous work did not focus on systems with NVRAM and underlying programming challenges. In addition to the background information on the FBM architecture in this section, we present specific caching policies and an automatic memory management scheme suitable for both transient and persistent memory in the context of FBM architecture.

A. Fixed-size read only data chunks

Both persistent and transient data is organized as a fixed size block, called ***chunks***, where the size of a chunk is the same as the cache line size. For the purpose of this study, we assume a 128-byte chunk and cache lines of the same size. Each chunk has a unique identifier, called a ***handle***, associated with it. Throughout the life time of a chunk, a handles is unique to the chunk and valid across the system. This is in contrast with the virtual addressing in conventional systems, where virtual addresses are valid only for the life time of the process. A thread creates a chunk in a processor's private transient scratch pad memory, and only the creating thread can write to it until it is finalized.

A set of API allows programmers to explicitly allocate and finalize persistent or transient chunks. Once finalized, a 64-bit handle is assigned to the chunk, and the chunk subsequently becomes read-only. When a finalized chunk is modified by the creator thread or a different thread, a new resultant chunk with a new chunk handle is created. A 128-byte chunk is divided into 16 64-bit slots. Apart from a handle, each chunk also has a metadata associated with it. A chunk metadata constitutes of the following: 1) tag bits for each slot in the block indicating the type of information stored in the slot, 2) a reference count for the chunk (see § III-C), and 3) a bit indicating whether the chunk is transient or persistent. Each slot in a chunk can have one of the following tag bits:

HANDLE: the 64-bit value is a handle to another chunk,
DATA: the 64-bit value is non-handle data,
LDATA: 32-bit value stored as leftmost bits,
RDATA: 32-bit value stored as rightmost bits, and
UNDEF: the slot is unused

Read-only data chunks have immediate consequences to NVRAM programming. It makes cache coherence protocol unnecessary system-wide, leading to a scalable architecture for accommodating a large amount of NVRAM. Copies of read-only chunk referred by a specific handle may be cached in multiple memory locations across the system, but all copies will have the same content. The content of a new chunk is visible to another thread only when the creating thread finalizes and shares its handle. This simple fact simplifies writing parallel programs by eliminating the race condition between read and write access to a chunk. The use of globally valid chunk handle addresses the challenge of having valid persistent pointers in NVRAM across execution cycles. The read-only nature of the data chunks and the globally valid handle enable sharing of persistent data across processes without having to worry about virtual address collisions and data coherence.

B. Fully associative memory hierarchy

Each processor has a scratch pad memory and relies on a hierarchy of cache for fast access to primary memory (NVRAM + DRAM), with L1 and L2 caches being private to a processor. One or more processors may share an L3 cache. All levels of cache, along with the transient and persistent memories, are ***fully associative***. Given a valid handle to a chunk, an associative unit at each level of the memory hierarchy maps that handle to the physical location of the chunk at that level if the chunk is present. A read miss at a certain level in the memory hierarchy prompts the system to fetch the chunk from the next level and so on. Chunks actively participating in computations are closer to the processors. Chunks at a given level of cache are replaced using ***least recently used policy (LRU)***. As chunks are read-only, they need not be written back to main memory when evicted. As a part of finalization step, a chunk is written to L1 and all the way to the primary memory (DRAM or NVRAM depending on the type of a chunk). It is also cached at each level of the memory hierarchy during the process (***write through policy***). As a large amount of memory will be distributed across a number of memory banks, all last level caches (LLCs) and memory banks will be connected by a crossbar switch. When accessing a particular chunk, if there is a cache miss at the LLC, bits in the handle of the chunk indicates the memory bank to which the chunk originally belongs (allocated).

Given a universally valid handle and an associative unit at each level, CPU cycles are not wasted computing effective memory addresses for accessing persistent data. This is in contrast with the previous work on NVRAM programming which either rely on relative addressing within an NVRAM region (.e.g [11]) or on having NVRAM region mapped to the same starting address over every execution cycle for absolute addressing.

Failure consistency in the proposed memory architecture can be achieved at no additional cost. Updates to chunks are automatically atomic with respect to failure. If a failure occurs before a partially written chunk is finalized, such a chunk is only present in processor's private scratch-pad and its incomplete set of updates is not visible after a failure. If a handle is returned to the chunk after a successful finalization, the chunk is guaranteed to be visible after a failure in NVRAM due to write through policy. This architectural guarantee allows

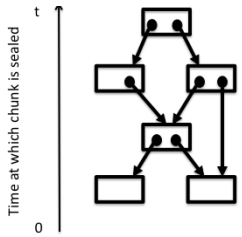


Fig. 3: Cycle-free heap in TBM

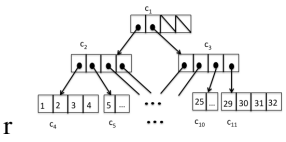


Fig. 4: Array as a tree of blocks

one to update data larger than a block in a fail-safe manner using a copy-on-write mechanism (see § III-D).

Write-through policy (WT) is usually considered inferior to write-back policy (WB), as WB potentially allows a number of updates to the same cache line be accumulated before the cache line is ultimately written back to the memory. In TBM architecture with read-only data chunks, we found this not to be the case. All modifications to the chunk in scratch-pad memory precedes the finalization step, and therefore, all updates to a chunk is accumulated and written all the way to the primary memory only once (as each chunk is finalized only once). Our experiments also confirm that WT in TBM architecture is no more expensive than WB. Furthermore, write-through in TBM architecture is cheaper than the write-back in conventional memory architecture in case of `NVRAM` programming, as there is no need to selectively flush modified cache lines (as is the case in WB in conventional memory system) in WT-enabled TBM.

Each memory bank is also backed by a slower archival storage (such as Flash SSD), which would act as an overflow swap space. Similar to the primary memory, data is stored as chunks with the same globally valid handle, with associative unit mapping the handle to the physical location of the chunk.

C. Hardware-level automatic memory management

In TBM, two or more chunks do not form a cycle. The handle to a chunk is only available after that chunk is finalized. Therefore, it can only be stored in other chunks that are finalized later in time as illustrated in fig. 3. Due to lack of cycles, a ***reference counting*** mechanism can be reliably used for an automatic garbage collection. Recall that a chunk's metadata indicates whether a chunk is persistent or transient. A handle to a persistent chunk can be stored in a transient chunk but not vice-versa. Transient chunks are all automatically reclaimed after a system restart rendering the transient handles stored in persistent chunks meaningless.

Each chunk has two types of reference count in its metadata: transient and persistent. For a transient chunk, the persistent reference count is always 0. A persistent chunk can have non-zero values for both types of reference counts. In cases of both, a transient and a persistent chunk, a chunk's transient reference (and not the persistent reference) is incremented to 1 when the chunk is finalized. When a persistent chunk, say p_1 becomes reachable from another persistent chunk, say p_2, the persistent reference count for p_1 is incremented by 1. Likewise, when a chunk (persistent or transient) is shared with another thread, or a reference to the chunk is stored in another transient chunk, its transient reference count increases. Recall that a chunk is private to a thread until it is shared with other threads by providing the handle. After the last reference to a chunk within a thread, its transient reference count is decremented. When the chunk's total reference count (sum of transient and persistent) is zero, the chunk is reclaimed, and its handle reused. For each handle stored in a chunk being reclaimed, the corresponding chunk's persistent or transient reference count is decremented depending on whether the reclaimed chunk is transient or persistent, and so on. Reference counted garbage collection can be automatically done and need not be in the critical path of program execution. A separate dedicated hardware thread is expected to perform the garbage collection work.

After a system restart, all transient chunks are reclaimed. Additionally, a transient reference count for each persistent chunk is reset to 0. Any persistent chunk with no persistent reference is then reclaimed. This prevents any persistent memory leaks due to a failure. Reconsider a scenario described in § II-D, where a failure occurs right after a persistent memory allocation. Under the scheme presented here, such chunks will only have a transient reference count which will be zeroed after a failure, and hence will be reclaimed.

All persistent data structures need an entry point, which we refer to as the persistent root. TBM API provides a method to specify a persistent chunk to be a persistent root. Specifying it as a persistent root increases its persistent count by 1, which ensures that the chunk and all other persistent chunks reachable from this chunk survives a garbage collection after system restart.

D. Storing data larger than a chunk

In TBM, all chunks are of a fixed size. Data that does not fit within a single chunk can be stored as a linked-list of chunks or more effectively as a multi-way tree of chunks. Fig.4 shows an array of 32 elements being stored as a tree of chunks. In this case, a persistent chunk c_1 can be considered the persistent root. Non-leaf chunks c_2 and c_3 store handles to the next level chunks, while leaf chunks c_4 - c_{11} stores elements of the array.

Suppose the 2nd and 31st elements in the array are to be modified atomically w.r.t failure. First a new leaf chunks in place of c_4 and c_{11} are created with new values for 2nd and 31st elements respectively, followed by a new non-leaf chunks in place of c_2 and c_3 containing handle to new leaf chunks and finally a new root chunk. If a failure occurs before a handle to the new root is established. The old tree remains intact, while any persistent chunks allocated before the failure is reclaimed

at the restart. Hence, a fail-safe update to the array (with all-or-nothing behavior) is guaranteed in this case at no additional programming or performance cost.

IV. Library-based emulator

Provided that the tree-based memory (TBM) architecture is a drastic departure from a conventional architecture, we found most of the simulators available today unsuitable for studying such a memory architecture. The absence of compiler support for TBM architecture only compounds the problem. Often times the resource intensive nature of building a cycle-accurate simulator and compiler support from scratch thwarts even the cursory exploration of promising unconventional architectures especially in the limited budget settings like the academia. For this reason, we developed a library based emulator for studying TBM in the context of NVRAM. Our emulator was helpful in demonstrating that a TBM is a viable architecture for accommodating NVRAM.

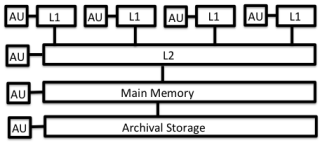


Fig. 5: Emulated TBM architecture

We implemented the emulator in Habanero Java(HJ) [14]. Fig.5 shows the emulated TBM architecture. It emulates a single socket of the TBM architecture, with up to 8 (processor private) L1 cache, a shared L2 cache, a main memory and the archival storage. We do not emulate the processors, registers or scratchpad memory. The programs are written in HJ and HJ runtime generates the task schedule. The HJ runtime worker threads are uniquely assigned to L1 caches being emulated. The emulator provides a set of ***get*** and ***put*** methods as API to store and retrieve chunks in the emulated memory architecture. All chunk transfers for a particular worker thread are then carried out through the assigned L1 cache. The *put* methods receive data to be stored in a chunk as well as the metadata information and return a handle to the chunk. The put method essentially writes a new chunk to L1 cache and finalizes it. Likewise, the get methods receive a handle to the chunk and return the chunk as well as the metadata associated with it.

The size of memory at each level of the hierarchy is configurable. The caching policy implemented is the same as described in § III-B. The emulated main memory comprises of NVRAM and DRAM. The chunks are offered to L1 cache via the same put method, but is either written to DRAM or NVRAM based on whether the chunk is persistent or transient as indicated by its metadata bits. The emulator records the total bytes read and written through each L1 cache, the amount of data transferred across each levels of memory hierarchy and reports the statistics at the end of the program execution. Implementation of automatic garbage collection has been left as future work.

V. Programmability of tree based memory

Task-structured parallel programming language (TSPL) such as Habanero Java (HJ) [14] provides a suitable programming approach for tree-based memory systems. Our experiments confirm that TBM architecture combined with HJ leads to a scalable and programmable NVRAM system. Based on X10 programming language, HJ is an extension of Java. The *async* and *finish* keywords represent the fork and join points in HJ respectively. An HJ program can be represented as a tree where nodes represent lightweight dynamic tasks, and edges represent the dependence between tasks. A computation graph for a well-written program using TSPL [14] can have roughly one-to-one correspondence with the chunks in a chunk tree representing the data to be processed. Furthermore, a lightweight task, along with the data chunk related to that task, can be easily migrated to the desired processor for proper work scheduling and load balancing.

For instance, to increment in parallel each element in the array in figure 4, the main task t_1 corresponding to the chunk c_1 may spawn and wait for two tasks t_2 and t_3 corresponding to chunks c_2 and c_3 respectively. Tasks t_2 and t_3 may further spawn and wait for a task for each leaf node. Each leaf task returns a handle for a new resultant leaf chunk to its parent task. Parent tasks at each level accumulate returned handles into a chunk and finally the main task returns a new root handle for the new tree.

A. In memory key-value store

We implemented the B-tree based in-memory persistent key-value store. We treated each chunk as a node in a B-tree. For the purpose of convenience, we assumed each key and each value to be 32-bit. It is possible to have arbitrary length keys and values (stored similarly as an array, see § III-D). In our experience, no significant modification was necessary to the textbook versions of B-tree related algorithms [15]. The B-tree nodes correspond to chunks in the emulator.

B. Matrix multiplication

Matrix multiplication is an important part of any linear algebra package. We implemented standard and recursive tile-based matrix multiplication [16] in HJ targeting TBM emulator. The core algorithm remained unaltered. The layout of the matrix in TBM required some effort. However, tree-based and tile-based recursive layout of matrix in conventional memory architecture for high cache locality and high performance have been extensively studied [16], [17]. The size of a base tile in our case matched the size of a chunk. We then used various layout based on space filling curves, namely U,X,Z,Gray)-Morton and Hilbert, for cache-friendly arrangement of tiles (as leaf nodes) in the tree-based representation of matrix.

In our experience, the algorithmic implementation for TBM is the same as equivalent fork-join parallel programs for conventional memory architecture. Some effort is required for matrix layout in TBM. On the other hand, tree-based layout is a more intuitive and a more efficient match for task-structured parallel programming paradigm. Furthermore, we believe that

much of the TBM layout details can be automatically handled by an appropriate compiler in the future.

VI. Related Work

Work on Fresh Breeze memory architecture [10] is most closely related to our work. HICAMP [13] is another memory architecture that is based on read-only data chunks. Both of these line of work is not pursued with next-generation NVRAM as a focus. As a result, these previous work on tree based memory architecture lack the assessment of challenges posed by future NVRAM technologies and the necessary insight into how to deal with the challenges such as failure induced data inconsistencies and memory leaks.

To make NVRAM programming more accessible, previous works such as Atlas [6], Mnemosyne [7], NV-Heaps [8] and Pmemio [11] has entirely focused on language extension and compiler support as opposed to the holistic approach of considering together the memory architecture and a fitting programming approach. As a result, much of the previous work did not address many of the important NVRAM-related programming challenges discussed in § II, such as cache coherence, persistent memory addressing, persistent memory garbage collection, and scalability. Other work on NVRAM such as Aerie [18] has focused on using NVRAM as an enhancement to file-system, rather than memory for general-purpose in-place persistence programming.

VII. Evaluation

We evaluated the proposed tree-based memory (TBM) architecture by collecting statistics on data transfers among various memory hierarchies using the emulator described in § IV. To study its characteristic difference in memory performance compared to conventional architecture, we extended and used MESI coherence protocol cache emulator, MultiCacheSim [19]. It performs PIN-based instrumentation of memory accesses and simulates assigned number of L1 cache and assigns memory accesses to the L1 caches based on the id of the thread that originated the memory access. Specifically, we extended MultiCacheSim to emulate cache line flushes. Although we studied data transfers at each level of the memory for TBM, we only present here the L1 cache performance due to lack of space. More detailed experimental results can be found in [20]. We set L1 cache size for both emulators to be 32 KB. In MultiCacheSim, the L1 cache is set to have 64-byte cache line size, with 8-way set associativity, LRU replacement and write back policy. This setting resembles L1 caches in most common conventional memory architecture. In MultiCacheSim, a cache line flush command resembles the Intel x86 functional behavior [21], i.e. it evicts and invalidates a cache line.

A. Transient memory programs

We first used transient implementation of matrix multiplication algorithms and layouts described in section § V-B to study and compare TBM with conventional architecture . Although detailed study can be found in [20], we only present results for standard matrix multiplication using Hilbert layout. We implemented a conventional memory version of this algorithm and layout. In conventional memory version, base tiles' relative postion are determined by the Hilbert layout function, but all tiles are laid out in one contiguous virtual address space. For fair comparison of conventional version with TBM emulation, we only emulated the memory access to the operand and resultant matrix elements using address filtering. So cache performance for both architectures are purely for matrix data.

Fig. 6a shows that TBM incurs comparatively larger amount (in bytes) of reads and writes compared to conventional architecture due to extra non-leaf nodes in tree-based representation. However, TBM enjoys significantly lower cache miss rate as seen in fig. 6b. Also, the total number of lines written out from L1 to L2 is lower for TBM, and most importantly remains constant w.r.t to the number of L1 caches as presented in fig. 6c. The primary reason for this is the lack of cache line ping-ponging in TBM. Conventional architectures suffer from this phenomenon due to more than one CPU needing to read or write to a cache line that exists in a modified state in another CPU's private cache. This is also the reason for an increase in coherence traffic along with the increasing number of L1 caches as shown in fig. 6d.

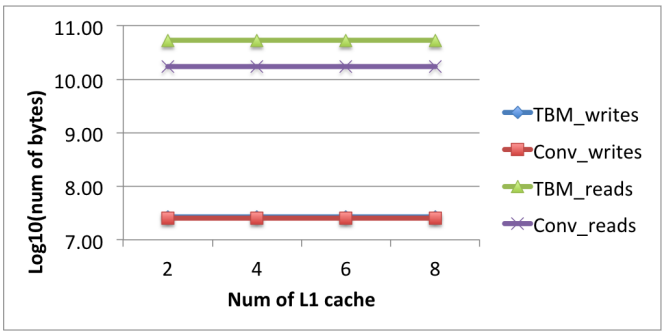
B. Persistent memory programs

We used in-memory persistent key-value store described in § V-A to compare the persistent memory program cache performance between TBM and conventional memory architecture. We compare the the performance of our implementation of key-value store (see § V-A) running on TBM emulator with the persistent in-memory version of MDB [22] implemented using Atlas [6]. MDB is a B-tree based memory-mapped commercially used OpenLdap backend database. Atlas uses transactional logs and cache line flushes to guarantee fail-safe updates to persistent data in MDB. For emulation purpose, we captured memory accesses to the B-tree in both architectures and additional transactional logs written by Atlas for failure atomicity support in conventional architecture.

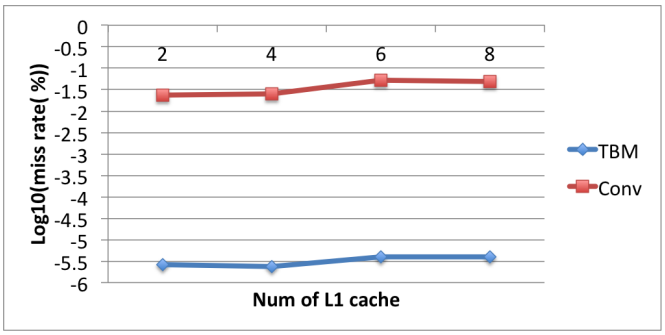
While our implementation of key-value store is purely copy-on-write, Atlas enabled MDB uses combination of copy-on-write and transactional logs for fail-safe guarantee. Due to this extra logs written in Atlas enabled MDB, TBM version performs lesser amount of total writes than the Atlas-enabled MDB as seen in fig. 7a. Furthemore, fig. 7b and fig. 7c show that conventional version incurs large L1 cache miss rate, and larger number of cache lines being written out of L1 cache to L2 respectively. This is primarily due to cache line flushes issued by Atlas-enabled MDB to guarantee visibility of updates in NVRAM. This step is unnecessary in our TBM architecture due to write through policy. Finally, the coherence traffic seen in fig. 7d is also absent in TBM.

References

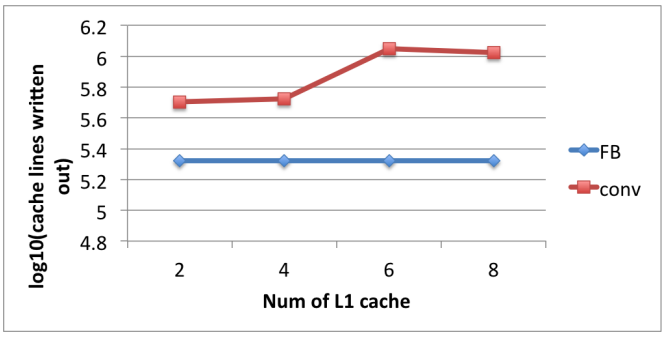
[1] *Process Integration, Devices and Structures*, 2007. [Online]. Available: http://goo.gl/rzDdQM



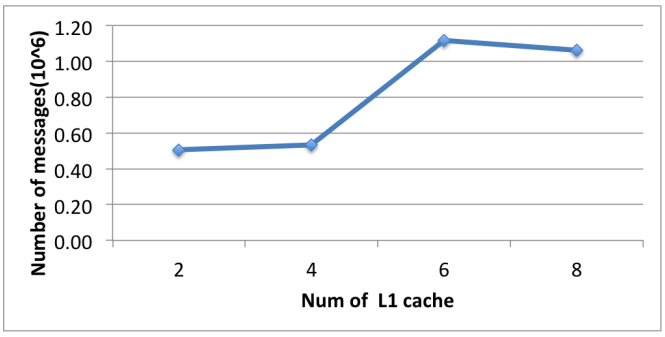
(a) Total reads and writes to L1



(b) L1 miss rate



(c) Total transfers from L1 to L2



(d) Total coherence traffic

Fig. 6: Matrix multiplication. Matrix size = 1024×1024 64-bit integers. TBM=tree based memory, Conv=conventional arch.

[2] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. of the 36th ISCA*. ACM, 2009, pp. 2–13.

[3] D. B. Strukov and et. al., "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[4] Intel, *Non-Volatile Memory*, http://goo.gl/uOfZHT, retrieved Nov. 2015.

[5] J. Edge, *A look at The Machine*, [Online; posted 26-August-2015]. [Online]. Available: https://lwn.net/

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the OOSPLA'14*. ACM, 2014, pp. 433–452.

[7] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 91–104.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 105–118.

[9] A. Dearle and et. al., "Orthogonal persistence revisited," in *Proceedings of the Second International Conference on Object Databases*. Springer-Verlag, 2010, pp. 1–22.

[10] J. B. Dennis, "Fresh breeze: A multiprocessor chip architecture guided by modular programming principles," *SIGARCH Comput. Archit. News*, vol. 31, no. 1, pp. 7–15, Mar. 2003.

[11] *Pmem.io: Persistent Memory Programming*. [Online]. Available: http://pmem.io/

[12] Y. Fu and et. al., "Coherence domain restriction on large scale systems," in *Proceedings of the 48th Int'l Symp. on Microarchitecture*. ACM, 2015, pp. 686–698.

[13] D. Cheriton and et. al., "Hicamp: Architectural support for efficient concurrency-safe shared structured data access," in *Proceedings of the 17th ASPLOS*. ACM, 2012, pp. 287–300.

[14] V. Cavé and et. al., "Habanero-java: The new adventures of old x10," in *Proceedings of the 9th PPPJ*. ACM, 2011, pp. 51–61.

[15] T. H. Cormen and et. al., *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[16] S. Chatterjee and et. al., "Recursive array layouts and fast parallel matrix multiplication," in *Proceedings of the 11th SPAA*. New York, NY, USA: ACM, 1999, pp. 222–231.

[17] P. Gottschling and et. al., "Representation-transparent matrix algorithms with scalable performance," in *Proceedings of the 21st Annual Int'l Conf. on Supercomputing*. ACM, 2007, pp. 116–125.

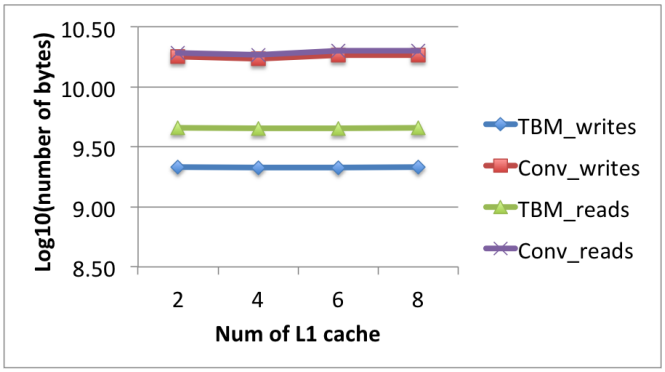
[18] H. Volos and et. al., "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the 9th Eurosys*. ACM, 2014, pp. 14:1–14:14.

[19] B. Lucia, *MulticacheSim*, https://github.com/blucia0a/MultiCacheSim.

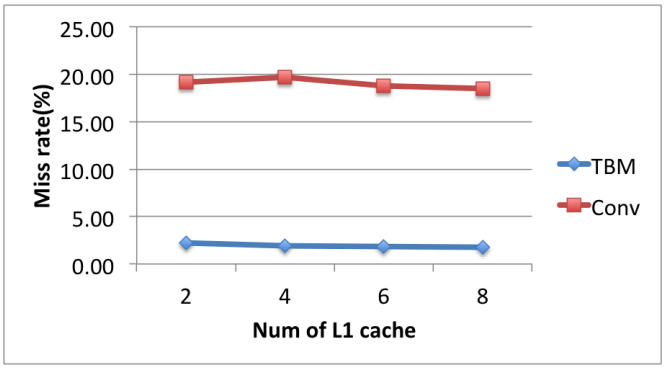
[20] K. Bhandari, "Evaluating the programmability and scalability of memory hierarchies with read-only data blocks," Master's thesis, Rice University, Houston, Texas, 4 2015.

[21] Intel Corp., *Intel64 and IA-32 Architectures Software Developers Manuals Combined*. [Online]. Available: http://goo.gl/B1tTk

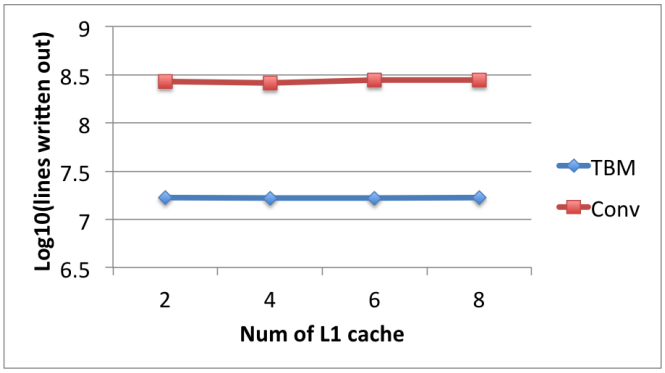
[22] H. Chu, "Mdb: A memory-mapped database and backend for openldap," *3rd Int'l Conf. on LDAP(LDAPCon)*, Oct. 2011.



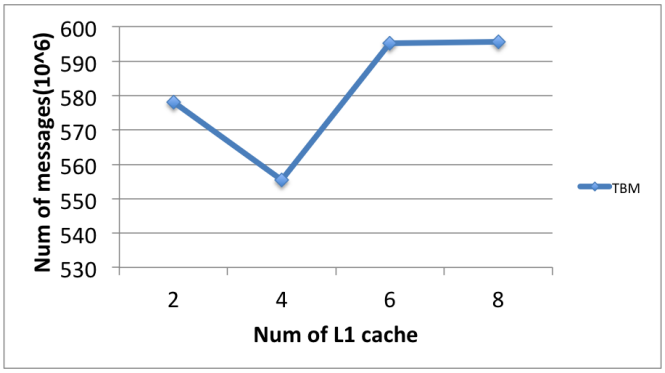
(a) Total reads and writes to L1



(b) L1 miss rate



(c) Total transfers from L1 to L2



(d) Total coherence traffic

Fig. 7: B-tree based key-value store. Insert/search size = 2,097,151 32-bit random keys with 32-bit corresponding values. TBM=tree based memory, Conv=conventional arch.