A Survey of Programming Interface For Persistent Memory

ABSTRACT

Non volatile memory provides great advantage but also challenge compared to conventional memory technologies. In this survey we investigate the challenge of persistent programming models and their interface design. We classify and analyze the common errors in persistent memory applications. We believe this survey will motivate more future research in the field for persistent memory software and testing framework. This survey

1. INTRODUCTION

STT-RAM [1] [2], [3] , and ReRAM [4] technologies introduces a new *persistent memory* concept in the memory and storage stack. Persistent memory allows fast byte-addressable access like main memory, and also inherits the data persistence from the storage hardware like disk. Blurring the boundary between DRAM and storage, persistent memory saves the users' work for (de)serialization and also the overhead in traditional data storage. That's why persistent memory is also called storage class memory (SCM).

The non-volatility property of persistent memory makes the traditional applications like database, programming language run-time and file system are no longer suitable for persistent memory. First, PM applications store data directly in persistence region in addition to accessing data through more traditional block-based filesystem and database interfaces, we also need to consider the dependency and performance between operating system, hardware with the PM application to achieve better performance and better management, like in heterogeneous memory systems.

Second, PM applications have to be crash consistent. Volatile programming don't have issues like that, in PM applications, programmers have to carefully reason about what struct to store and their order. At low level, these properties are ensured by two primitives, (1) clflush, literally cache line flush back, to explicitly write data back fro m the processor cache to the persistent memory. (2) fense, enforcing ordering between writes to dependent structures [15]. The set of writes between ordering points constitute an epoch [5] [6] and can reach PM in any order, and between epochs, the store operations are strictly ordered.

Software can implement persistent transactions with ACID semantics using the ordering guarantees of epochs. But there are still issues with raw transactions mechanism as real world data structures are highly related, we may need highly crash consistency model to ensure this.

Contributions: In this paper, we present a survey that:

- investigates the mainstream persistent memory applications and their interface(APIs)
- investigates the theory foundation for crash consistency and their application in software design.
- propose an application level hierarchy to classify different sources of bugs, which could be combined to testing framework by stacked checkers
- describe the bugs in these class and analyze their relationship with interface design and programming habits

2. AN OVERVIEW OF PM PROGRAMMING MODELS AND APIS

What is a better interface? To guarantee consistency and durability, raw non-volatile structures must meet a bunch of challenges which do not appear in volatile memory. Like dangling pointers, which will not be retrieved after reboot, will be serious issue for persistent memory, multiple free()s, memory leaks and locking errors. Moreover, there are many implicit programming errors, like pointers from non-volatile data structures into volatile memory are unsafe, as when the program crashes, the non-volatile pointer still exists, but it becomes a null pointer. Trusting the users to reason about all and write robust and safe code is dangerous and highly inefficient. So a good interface must be:

- Easy to understand, users can draw experiences from former volatile memory programming
- Expose less internal details about PM models, then the users don't need to reason much about the hardware.
- Less programming invariants or rules, that programming will be less error-prone.

The naive approach is exposing NVMs as raw storage, which is mapping PM into virtual address space and managing non-volatile memory using volatile memory model. Although this method is easy to understand and familiar to volatile memory programmers, it is error-prone and hard to work with. NV-heaps [7] provides a persistent object system for providing transactional semantics and a robust persistence model. To ensure crash consistency, the system provide safety mechanism in garbage collection, pointer safety and easy-to transactions, while the performance is close to raw NVM. Moreover, the NV-heaps is built on existing file systems and tools, which require less modification to the operating system thus very user-friendly.

Mnemosyne [8], an interface for persistent programming, provides another way to integrate with the traditional software stack. The purpose of Mnemosyne is to reduce the store overhead in persistence memory. In Mnemosyne, users write data to storage directly rather than writing the pages back through filesystem. The ordering of writes is controlled by software using PM primitives. The system ensure consistency through a lightweight transaction mechanism, which has more flexible than NV-heaps.

When using low-level primitives, omission of even a single flush, fense operation can result in errors. As undo/redo logs are very common operations in persistent memory, the *transaction* concept is proposed to work as a more organized way to ensure crash consistency and avoid writing the same functional code over and over again in low-level library. Marathe et al. notices the transaction implementation is highly related to the specific run-time system and lack optimization [9].The paper presents three basic transaction run time and a memory management algorithm to optimize the number of persistent barrier, which is heavy overhead in run time.

As for concurrency, Kolli et al. proposes a relaxed crash consistency model to optimize the transactions in multi-threading [10]. The system makes use of a deferred commit transactions to minimize persist dependencies and achieve high performance. The work also presents a performance analysis framework for transactions.

When integrated with programming language and its object system, the interaction of persistent memory with object-oriented programming (OOP) need to be considered. OOP complicates the allocations and deallocation behavior, makes the data access become encapsulated and segmented. The inheritance and composition makes the data store implicit, either in non-volatile and volatile memory. Ren et al. design a new dichotomy design pattern to separate allocation and access [11]. In addition, a programming language with managed run-time system makes the problem even harder. This additional layer of abstraction brought the JAVA virtual machine complicate the persistence management. Wu et al. presents Espresso [12], a general persistent heap to achieve object persistence. A new abstraction called Persistent Java Object is also designed to provide an easy-to-use programming model, while being compatible to existing data structures in Java programs.

Combining hardware design with the software also attracts many researchers. Narayanan et al. [13] works on a full non-volatile memory system. Their approach is to flush the CPU registers and cache data to the persistent memory only on a failure by hardware. On the reboot, the OS and application are working as a whole system log and then are recovered. But the systems lack extensibility and scalability.

Similarly, Zhao et al. takes a milder approach by designing a non-volatile last level cache and main memory to construct a persistent memory hierarchy. In this system, the newly updated data is persisted in cache line and the old data is stored in main memory. The system allows persistent in-place updates without logging.

There are also testing framework that aimed at rectifying users' programming error by testing. The NVL-C [14] proposes a programming model, its prototype and a novel static analyses and transformations that test the program at compile

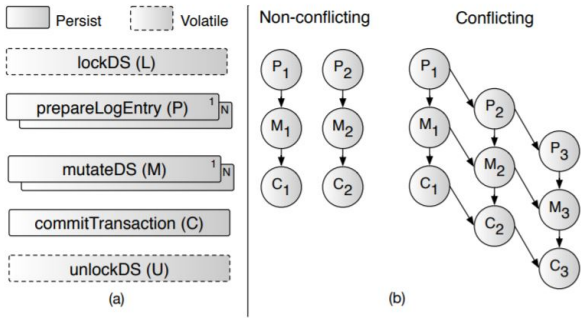

Figure 1. (a) Steps in an undo transaction (b) Persist dependencies in a transaction sequence

time. But static analysis is weak in dynamically allocated structures, so dynamic analysis tools like Yat [15] appears. Yat took an exhaustive searching approach to find every possible execution in run-time and test the memory snapshots with the checkers. Liu et al. proposes PMTest [16], a test framework that provides both low-level and high-level checkers to test a wide range of persistent memory. It is easily integrated with other software stacks and requires writing little glue code to reuse the primitives of the test framework.

3. CRASH CONSISTENCY MODEL

programming in persistent memory introduces the basic idea of crash consistency, which is durability and ordering. Simply write to persist memory don't literally persist them. To solve the program, the clwb operation is provided for durability to make sure that the cache line is written back and thus we can ensure the data to be persisted. Enforcing the ordering is another fundamental function for crash consistency software (CCS). An ordering guarantee from PM system is the fense operation. For example, the common undo log requires the log to be persist before updating the transaction. These strict ordering relationship requires a semantic order constraints for CCS.

But considering multiprocessors, programming becomes more tricky. Compared to traditional memory model, NVRAM requires a memory model because of the crash consistency. Moreover, in performance for example, NVRAM technologies exhibit asymmetric read and write latencies. It is a completely different scenario for application design and programming primitives. Models about memory consistency has been explored very completely [17]. It allows programmer to reason about the visible order of loads and stores among threads. Sequential Consistency(SC), requires that all loads and stores occur as some interleaving of program orders of each thread, often requires delays. To achieve better performance, relaxed consistency models explicitly allow certain memory operation to reorder. In a thread's perspective, the ordering from other threads are therefore different. But in memory persistency, the recovery mechanisms define specific orders on persists. Like in figure2, the store a must be before store b as the semantics requirement. But if the former memory model reorders them, the persistent consistency is failed. The paper [6] regard failure as a recovery observer that atomically reads all of persistent memory at the moment of failure. Ordering constraints thus become ordering con-

straints on memory and persistent operations as viewed from the recovery observer. With this abstraction, we can borrow some tools from meory consistency – any two stores to the persistent memory may apply an ordering constraints, which is defined by the oberser. However, these consistency model in Figure 1 can be further relaxed in [10], which makes the multiprocessor memory application design more flexible, but also more error-prone. In this way, we define a new memory order by our model. The program execution basically defines an order. We separate the memory order of volatile memory and persistent memory and look at our implementation in [6].

3.1 Strict Persistency

Our first persistency model is strict persistency. Strict persistency is very easy to implement and simply greatly reasoning about persistent ordering by coupling persist dependences to the memory consistency model. **Definition** Strict persistency persist order observes all partial order implied by volatile memory order. Therefore, the program order is the persists order. We can make sure safety by these strict model. **Implementation** An obvious implementation is to stall the next operation util the afore-executed operation is done. Conventional prediction mechanisms may allow loads to speculatively reorder. Buffered strict persistency can be utilized tby serializing persists to a single ordered queue.

3.2 Relaxed Persistency

Strict persistency is safe but lack flexibility. Therefore, a relaxed persistency model is proposed to decouple memory consistency and persistency model, as they are actually separate and have gaps to achieve better performance. Relaxed persistency loosens persist ordering constraints by application-defined barrier to enforce the visibility of memory operation ordering with respect to other processors.Using the language of ordering constraints in [10]:

- L_a^i : A load from thread i to address a
- S_a^i : A store from thread i to address a
- M_a^i : A memory operation load/store from thread i to address a
- A $\leq_v$ B: A occurs no later than B in volatile memory order
- A $\leq_p$ B: A occurs no later than B in non-volatile memory order

We describe two relaxed persistency models.

Epoch persistency The epoch persistency model introduces the persist barrier. We denote the persist barriers issued by thread i as PB^i Under epoch persistency, any two memory accesses on the same thread that are separated by a persist barrier in volatile memory are ordered in persistent memory: $M_a^i \leq_v PB_a^i \leq_v M_b^i$ -> $M_a^i \leq_p M_b^i$

Strand Persistency Strand persistency divides program execution into strands. Those strands are logically independent segments that live in the same thread. New Strands event from thread i are denoted as NV^i. The new thread event clears all prior PMO constraints from prior instructions.Memory access within a strand are ordered using epoch persistency.

But in strand consistency, two memory accesses on the same thread separated by a barrier are ordered in persistent memory only if there is no intervening strand barrier.

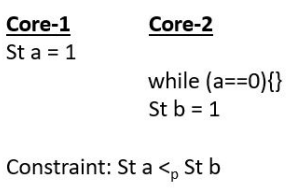

Figure 2. a,b=0 at initial

4. TAXONOMY OF API

There are now so many interfaces and their own coding rules, mechanisms and invariants. A review of different API is the first step to understand the programming errors. We summarize the existing persistent programming interface and classify them by control granularity. Fine-grained control is more difficult and requires the user to understand more about persistent memory, but can give experienced programmers more freedom in controlling the details and thus achieve high performance. Coarse-grained control is often for high-level application to write fast code and reduce the possibility of errors, but it is encapsulated as high level primitives which users have no control over internal operations.

4.1 Low-Level API

The low-level primitives fully expose the requirement of persistent memory to programmers, thus it is the most difficult API. Programmers have to carefully write every NVM access and reason about their orders. Like the Intel PMDK library, provides a libpmem library with a wrapper pmem_ersist() that is CLWB and FENSE. The libpmem also provides basic cache line flush back and persist barriers. Ren et al. uses a independent syntax to summarize this: __nv(variables ...) statements; ..., which means to persist these variables using these primitives [11].

4.2 Transaction API

To save the burden to write naive undo/redo code in every NVM access, a compiler can help to do the work to automatically convert users' connotation to the store inside a transaction. Mnemosyne[8] leverages this approach. In transaction API, a programming only need to specify a block of code and use the transaction macro, then every memory access will be taken care of by the transaction run-time. If we follow the low-level syntax, a transaction is like a __nvstatements;.., which don't need to specify the exact variables. Basically, a transaction is considered a atomics operation that could roll back/recover the update when crash happens.

4.3 Object-Level API

The low-level primitives and transactions are working as a data access. A system approach is to specify a consistency state, and by making every update persistence consistent, we make the object persistence consistence. The NVL-C [14] provides a prototype programming language, the two constraints are that programmers have to specify non volatile

```
static void
list_insert_inconsistent(struct list_root *root,
 node_id node,
 int value)
{
    struct list_node *new = NODE_PTR(root, node);

    new->next = root->head;
    pmem_persist(&new->next, sizeof(node));

    root->head = node;
    pmem_persist(&root->head, sizeof(root->head));

    new->value = value;
    pmem_persist(&new->value, sizeof(value));
}
```

Figure 3. PMDK code to insert a element to a linked list

pointers and all referenced objects must be placed in NVMM. The Intel PMDK supports p<> template pointer that take care of both the basic data types and complex user-defined objects. It also provides a smart pointer persistent_ptr<> that is compatible with modern C++. Other specific and specially designed objects protection mechanism like [18] [19] are internally protected, and the protection is transparent to users who never modify these data structures. A more adanced example is to help automate recognizing persistent objects. NVMOVE [20]make static analysis about a traditional application and find out which part needs to be persistent if transplanted into PM application.

4.4 Program-Level API

After object level protection, it's a natural idea to achieve whole program crash consistency protection. The ThyNVM [21] and whole system persistence (WSP) [13] are special architecture designs that can persist the whole space efficiently. WSP flushes volatile CPU states using residual energy on a power outage. ThyNVM frequently generates checkpoints during execution time. The program-level API can directly work with traditional memory code, and transparently guarantees crash consistency of NVMM data.

Based on the different level of programming API, we will describe the common programming bugs, misuse of the programming models.

5. TAXONOMY OF PROGRAMMING ERORS

5.1 data structure dependency

In figure2, the code performs the linked list insertion operation. As this is a low-level library, we need to understand the data structures and operations so that we can specify the order explicitly. Because writing undo log for all is not friendly to space usage. The operation is special in that, you need to update the new node and insert it into the linkedlist, or, when you insert the node to the linkedlist, you have to make sure the node is updated. That's the crash consistency for our data structures. This piece of code insert the null node before updating the value. If the system fails before updating, there will be error with the persisted linked list.

This kind of bugs have something to do with data dependency. In programming, our data often have their own dependency, that when we modify one of them, we need to consider

```
1  class Counters {
2    int *counts;
3   public:
4    Counters(int n) {
5      counts = nv_new int[n];
6    }
7    int Increase(int i) {
8      __nv (counts[i]) {
9        return ++counts[i];
10     }
11   }
12 };
13 class TimedCounters : public Counters {
14   time_t timestamp;
15  public:
16   TimedCounters(int n) : Counters(n) {
17     __nv (timestamp) {
18       time(&timestamp); // get time now
19     }
20   }
21   time_t GetTime() {
22     __nv (timestamp) {
23       return timestamp;
24     }
25   }
26 };
27 int main() {
28   TimedCounters *tc =
29       nv_new TimedCounters(1);
30   cout << "Since " << tc->GetTime() << ": ";
31   cout << tc->Increase(0) << endl;
32 }
```

Figure 4. A time counter

other data that depends on it. In object-oriented design, we design clear and decoupled interface to avoid data entanglement. But in low-level memory model, the reorder of memory writes break the boundary of our interface. Unexpected failure will happen due to the fact that data dependency require the updating to be atomic, or have a specified order.

The data dependency is very common in low-level library, as in traditional programming model, we do not predict a crash everywhere in the program and thus we don't pay much attention to the order of writes to ensure a safe update.

5.2 OO design dependency

The class Counter is defined to count the events through its function Increase(). It use an array of integers: counts so that threads can be statically assigned to different integers to avoid contention. We define another TimedCounter class to inherit the Counters.

The timedCounter initialize its timestamp with the initialization time. Each increase adds one count. The design follow the encapsulation of OOP. Moreover, the Timedcounter is allocated in non-volatile memory(line 29) and all accesses are protected (Line17).

The program seems plausible, but there is one implicit bug in it. Timedcounter inherits the pointer counts from Counters, and the object is allocated in NVM. Therefore, access to the pointer should be protected with atomic protection. But when we allocate the Counters in Line 5, the counts are not initialized with safety. So the program is buggy due to a implicit data inheritance. These kinds of bugs are also very hard to find.

Figure 5. Another timed counter

Consider another program that allocate the TimedCounter in volatile memory, but the ptr_timestamp still points to the NVM, while the ptr_timestamp lives in volatile memory with TimedCounter. But line 16 still protect the pointer, which is unnecessary and affect the performance. In conclusion,

```
class Counters {
  int *counts;
 public:
  Counters(int n) {
    counts = nv_new int[n];
  }
  int Increase(int i) {
    __nv (counts[i]) {
      return ++counts[i];
    }
  }
};
class TimedCounters : public Counters {
  time_t *ptr_timestamp; // a pointer
 public:
  TimedCounters(int n) : Counters(n) {
    __nv (ptr_timestamp, *ptr_timestamp) {
      ptr_timestamp = nv_new time_t;
      time(ptr_timestamp); // get time now
    }
  }
  time_t GetTime() {
    __nv (*ptr_timestamp) {
      return *ptr_timestamp;
    }
  }
};
int main() {
  TimedCounters tc(1);
  cout << "Since " << tc.GetTime() << ": ";
  cout << tc.Increase(0) << endl;
}
```

Figure 4. Another time counter

mismatch of data placement(in DRAM or NVM) and persistent primitives causes many bugs in persistent programming model.

5.3 careless mistake

Last but not least, there are also so many careless mistake made by programmers like forgetting to set transactions, forget the flush back cache line. Especially in complex data structures, these kind of bugs are inevitable even in non-persistent memory programming. Using high-level abstraction could reduce the possibility of such mistakes.

6. CONCLUSION AND FUTURE WORK

In this survey, we investigate the PM application and theory for crash consistence. We realize that API designs are really important for users to write safe and high performance code. We classify different granularity of control to the persistent memory. Digging into the bugs and their source, we conclude that:

In the programmers' perspective, API should be designed with less program invariants and be closed to traditional programming. For example, rules like flush back cacheline after writes are extremely error prone. In addition, using a familiar API design like the object protection in [12] is very easy to understand and thus reduce some unexpected mistakes by users.

In the software engineering's perspective, API should expose as little internal details as possible. The designer cannot assume that the users to be familiar with persistent memory. The programming rules should be based on a high level concept like a transaction, or even a program-protection system.

There are also another approach that can reduce the programming errors like PMTest [16]. The paper has already made a unified framework about persistent programming checkers. Future work based on that may be in:

- We may want to try a learning approach to do automatic bug detection based on static analysis.
- Considering the vast amounts of PM application, designing a persistent memory-specific language to unify the different programming models is necessary.

7. REFERENCES

[1] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 480–617, Feb 2007.

[2] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, April 2013.

[3] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, (New York, NY, USA), pp. 1:1–1:17, ACM, 2013.

[4] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, pp. 2237–2251, Dec 2010.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[6] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 265–276, IEEE Press, 2014.

[7] 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," *SIGPLAN Not.*, vol. 46, pp. 105–118, Mar. 2011.

[8] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGPLAN Not.*, vol. 47, pp. 91–104, Mar. 2011.

[9] V. J. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, "Persistent memory transactions," *CoRR*, vol. abs/1804.00701, 2018.

[10] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *SIGPLAN Not.*, vol. 51, pp. 399–411, Mar. 2016.

[11] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for non-volatile main memory is hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, (New York, NY, USA), pp. 13:1–13:8, ACM, 2017.

[12] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing java for more non-volatility with non-volatile memory," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 70–83, ACM, 2018.

[13] D. Narayanan and O. Hodson, "Whole-system persistence," *SIGPLAN Not.*, vol. 47, pp. 401–410, Mar. 2012.

[14] J. E. Denny, S. Lee, and J. S. Vetter, "Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, (New York, NY, USA), pp. 125–136, ACM, 2016.

[15] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson, "Yat: A validation framework for persistent memory software," in *USENIX Annual Technical Conference*, 2014.

[16] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," ASPLOS '18, ACM, 2019.

[17] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, pp. 66–76, Dec 1996.

[18] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2011.

[19] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. J. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST*, 2015.

[20] H. Chauhan, I. Calciu, V. Chidambaram, E. Schkufza, O. Mutlu, and P. Subrahmanyam, "Nvmove: Helping programmers move to byte-based persistence," in *INFLOW@OSDI*, 2016.

[21] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 672–685, Dec 2015.