Quartz: A Lightweight Performance Emulator for Persistent Memory Software

Haris Volos[1], Guilherme Magalhaes[2], Ludmila Cherkasova[1], and Jun Li[1]
[1]Hewlett Packard Labs, [2]Hewlett Packard Enterprise
{haris.volos, guilherme.magalhaes, lucy.cherkasova, jun.li}@hpe.com

ABSTRACT

Next-generation non-volatile memory (NVM) technologies, such as phase-change memory and memristors, can enable computer systems infrastructure to continue keeping up with the voracious appetite of data-centric applications for large, cheap, and fast storage. Persistent memory has emerged as a promising approach to accessing emerging byte-addressable non-volatile memory through processor load/store instructions. Due to lack of commercially available NVM, system software researchers have mainly relied on emulation to model persistent memory performance. However, existing emulation approaches are either too simplistic, or too slow to emulate large-scale workloads, or require special hardware. To fill this gap and encourage wider adoption of persistent memory, we developed a performance emulator for persistent memory, called Quartz. Quartz enables an efficient emulation of a wide range of NVM latencies and bandwidth characteristics for performance evaluation of emerging byte-addressable NVMs and their impact on applications performance (without modifying or instrumenting their source code) by leveraging features available in commodity hardware. Our emulator is implemented on three latest Intel Xeon-based processor architectures: *Sandy Bridge, Ivy Bridge*, and *Haswell*. To assist researchers and engineers in evaluating design decisions with emerging NVMs, we extend Quartz for emulating the application execution on future systems with two types of memory: fast, regular volatile DRAM and slower persistent memory. We evaluate the effectiveness of our approach by using a set of specially designed memory-intensive benchmarks and real applications. The accuracy of the proposed approach is validated by running these programs both on our emulation platform and a multi-socket (NUMA) machine that can support a range of memory latencies. We show that Quartz can emulate a range of performance characteristics with low overhead and good accuracy (with emulation errors 0.2% - 9%).

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of Systems; D.2.6 [**Software**]: Programming Environments

Middleware'15, Dec. 7-11, 2015, Vancouver, Canada.
 ISBN 978-1-4503-3618-5/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814806.

General Terms

Measurement, Performance, Design

Keywords

Performance modeling, benchmarking, profiling, performance counters, memory throttling

1. INTRODUCTION

Persistent memory (PM) has been gaining momentum both in research [20, 19, 36, 30, 21, 29] and industry [13, 12] as a promising interface for accessing emerging byte-addressable non-volatile memory (NVM) technologies, such as phase-change memory and memristors. With persistent memory, NVM is directly attached to the memory bus and accessed through regular processor load/store instructions, thus avoiding the overheads of traditional block-based interfaces. Forward-looking projects like Berkeley's Firebox [17] and HP's The Machine [16] envision future scale-out machines that have enormous amount of non-volatile memories (NVMs). Efficient Big Data processing poses many challenging performance problems to solve. There are many open questions about possible system software design with NVMs such as:

- Shall we consider DRAM as a caching layer for NVM?
- Shall we build systems with two types of memory: DRAM (fast, small) and NVM (slow, large)?
- Given two memory types, how shall we design new applications to benefit from this memory arrangement and decide on the efficient data placement?
- How sensitive the applications are to different ranges of NVM access latency and bandwidth?

Apparently, many design and data placement decisions might depend on the *performance characteristics (latency and bandwidth)* of future NVMs. It is non-trivial to predict applications performance when it is executed on NVM with higher access latency. However, NVMs are not commercially available yet, and a few existing hardware prototypes [18, 21] have limited accessibility. Therefore, there is a high need for an emulation platform that mimics performance characteristics of different NVM technologies for assisting researchers in design of new software stacks for emerging NVMs and studying their performance on future workloads (without modifying or instrumenting the application source code).

Persistent memory performance is expected to be close to performance of main memory (*i.e.*, DRAM). However, its exact performance characteristics will depend on the underlying NVM technology and hardware. Due to lack of commercially available NVM, system software researchers have mainly relied on emulation to study how system software and algorithms would perform with different PM performance characteristics, such as bandwidth and latency. [20, 19, 36, 30, 21, 27]. Current emulation approaches

though suffer several limitations: cycle-accurate architecture simulators are too slow (four or more orders of magnitude slowdown) and do not scale to large workloads [20, 32], emulators based on special hardware limit adoption [21], and purely software-based approaches are either too simplistic and ignore cache eviction effects and memory-level parallelism [10, 36], or require separate offline analysis [30].

In this paper, we introduce a lightweight performance emulation platform for persistent memory, called Quartz[1]. Quartz focuses on modeling the primary **performance characteristics** of PM that contribute to application end-to-end performance, namely **latency** and **bandwidth**. We *do not target low-level hardware details*, such as specific NVM devices, processor, and memory system architecture, as these require heavyweight simulation techniques [31, 25]. Thus, we **are not** after an accurate simulation of NVM functionality and NVM features, but rather after emulating the NVM *performance characteristics*.

As a general design principle to achieve a low-overhead emulation at large scale without compromising accuracy, while encouraging wide adoption, our emulator utilizes hardware features available in commodity processors to slow down DRAM.

We take a two-pronged approach to emulating persistent-memory performance. We emulate bandwidth by utilizing the DRAM thermal control feature available in commodity processors [6, 24] to limit (throttle) available memory bandwidth similarly to other efforts [21]. Emulating latency is more challenging. Commodity hardware provides no direct control of memory latency, so we have to resort to software-based techniques. Software, however, introduces a very high overhead for slowing down each individual memory access. We instead *focus on modelling average application perceived latency* to be close to PM latency. The key idea is to dynamically inject software created delays to account for higher PM latency at boundaries of specially defined time intervals, called *epochs*. We derive delays based on an analytic model that leverages hardware performance counters to achieve low overhead and good accuracy.

Our earlier work-in-progress report [33] sketches the initial analytic model for single-threaded applications and provides preliminary evidence of its effectiveness. Here, we extend this initial model to cover multithreaded PM applications with inter-thread dependencies and communications, and present a complete implementation based on that model.

We implement our emulator in Linux as a combination of two modules:

- a *user-mode library* that dynamically attaches to a program to emulate PM;
- a kernel module that programs necessary hardware registers and performance counters.

The library dynamically attaches to a program and emulates persistent memory performance by slowing down DRAM based on our analytic model. It uses the kernel module to properly control hardware registers and performance counters as needed by the model. Our emulator is implemented on three latest Intel Xeon-based processor architectures: *Sandy Bridge, Ivy Bridge*, and *Haswell*. Moreover, we extend Quartz for emulating the application execution on future systems with two types of memory: fast, regular volatile DRAM and slower persistent memory.

We evaluate the accuracy of the proposed solution by using a set of specially designed microbenchmarks executed on three different hardware platforms. We show that Quartz can emulate a range of performance characteristics with low overhead and good accuracy (with emulation errors 0.2% - 9%). We demonstrate the utility of our emulator by studying the sensitivity of a key-value store and graph algorithm to PM performance.

The rest of this paper is organized as follows. Section 2 introduces the analytic memory model used in our emulation platform, including extensions to cover multithreaded applications. Section 3 presents a complete emulator implementation based on that model. Section 3.3 describes another Quartz extension for emulating the application execution on systems with two types of memory (DRAM and NVM). Section 4 describes the experimental setup and applications to evaluate the accuracy and effectiveness of our approach. Section 5 outlines related work. Section 6 discusses implementation challenges and future opportunities for Quartz. Finally, Section 7 presents conclusion and future work directions.

2. MEMORY PERFORMANCE MODEL

In this section, we discuss subtleties in emulating NVM using DRAM and the requirements for separately mimicking two performance characteristics of NVM when compared to DRAM: lower memory bandwidth and higher latency. To that extent, we propose separate memory bandwidth and memory latency performance models, and discuss their implementation based on commodity hardware.

2.1 Bandwidth Model

We emulate bandwidth by leveraging the DRAM thermal control feature available in commodity processors to limit available memory bandwidth similarly to other efforts [21]. Specifically, we utilize thermal control registers found in the integrated memory controller of modern Intel Xeon processors [7] to programmatically throttle DRAM bandwidth in a per channel basis.

Intel manuals [6] describe separate registers for *read* and *write* bandwidth throttling, which might provide separate knobs to independently control *read* and *write* memory bandwidth. This approach could provide additional advantages and flexibility: it enables modeling the asymmetric bandwidth specification of NVM, where generally the *read* bandwidth is greater than the *write* bandwidth[2].

2.2 Epoch-based Latency Emulation

As commodity hardware provides no means to effectively control memory latency, we employ a software-based solution for emulating PM latency. Software, however, introduces a very high overhead for slowing down each individual memory access. We instead focus on modelling average application perceived latency to be close to PM latency. The key idea is to dynamically inject software created delays to account for higher PM latency at boundaries of specially defined time intervals, called *epochs*, as shown in Figure 1. For a given epoch, our emulator observes application's compute and memory characteristics by reading hardware performance counters. Then, at the end of each epoch, Quartz calculates the required software delay using the collected memory metrics, it interrupts the application execution, and injects the delay by spinning for the required time. For a *single threaded application* the epoch creation is as simple as creating the *fixed-size* intervals.

A very *simple* model for computing the additional delay Δ_i for a given epoch i is to count the total number of memory references made in $Epoch_i$ and multiply it by the difference in NVM and DRAM latencies. But the point to note is that not all the memory references issued by the application are served by DRAM, because some of the references are served by the processor's private caches and/or shared last level cache. Therefore, we need to count

[1] Like quartz in electric oscillators, our emulator regulates time.

[2] In our experiments, we were able to throttle overall memory bandwidth. We have contacted Intel engineer with questions on how to make the separate registers for *read* and *write* bandwidth throttling work, and found out that these registers are not yet broadly available in many latest processors.

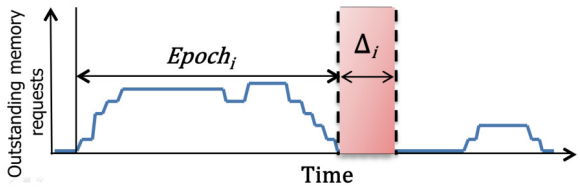


Figure 1: Our approach for emulating NVM latency by injecting software delays after each epoch.

only those memory references that *miss the caches* and are actually served from memory (M_i). So the additional delay for a particular epoch i can be defined as follows:

$$\Delta_i = M_i \cdot (NVM_{lat} - DRAM_{lat}) \quad (1)$$

where we use the following denotations:

- Δ_i - software delay injected at the end of epoch i.
- M_i - the total number of memory references going to the memory system in epoch i.
- NVM_{lat} - the average NVM access latency (in ns).
- $DRAM_{lat}$ - the average DRAM access latency (in ns).

This *simple* model works well *iff* all the memory references are issued serially to memory one after another. Figure 2 shows pictorially different memory reference processing patterns, that require injecting different delays for emulating a slower NVM latency. We can observe that $Load_A$, $Load_B$, and $Load_C$ are issued serially in $Epoch_1$, and therefore, Eq. 1 correctly models the additional delay for this epoch as shown in Figure 2 and $Epoch_1$.

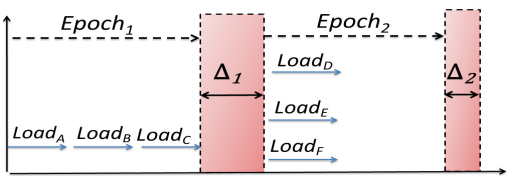


Figure 2: Impact of memory level parallelism on calculating the software delay injected at the end of each epoch.

Several features of modern processors make this assumption invalid. Memory is typically write-back cacheable: a write is cached and the data it writes goes out to memory when the cache finally evicts it, while a read may hit in the processor cache and avoid the long trip to memory. Moreover, hardware memory prefetching may further reduce the number of memory references a processor stall waits for. Finally, modern processors may issue multiple memory requests in parallel, known as *memory level parallelism (MLP)*, to overlap requests and hide memory latency. As shown in Figure 2 and $Epoch_2$ with memory references $Load_D$, $Load_E$, and $Load_F$ issued in parallel, the *simple* model over-estimates the additional delay by a factor of 3, because of not considering the impact of MLP during memory reference processing (MLP=3 in this epoch). Therefore, to account for MLP we should approximate the average number of *serial* memory accesses for computing the additional delay in a given epoch i.

To our rescue, most recent processors provide hardware performance counters to measure the number of cycles a processor stalls when serving LOAD memory requests. Memory stall cycles (denoted as *LDM_STALL*) naturally capture memory-level parallelism as other memory requests that are served in parallel to an outstanding request do not contribute to stall cycles. Thus, dividing memory stall cycles by the average memory latency gives us the average number of serial memory accesses. Then the additional delay to inject after $Epoch_i$ is defined as follows:

$$\Delta_i = \frac{LDM\_STALL_i}{DRAM_{lat}} \cdot (NVM_{lat} - DRAM_{lat}) \quad (2)$$

where LDM_STALL_i is the total number of processor stall cycles caused by serving memory requests in epoch i as estimated via hardware performance counters.

2.3 Latency Emulation for Multithreaded Applications

The epoch-based latency emulation (with statically configured epochs) works well for single-threaded applications. Figure 3 presents how the emulation works in presence of two independent threads, which run asynchronously. In this scenario, there are no any dependencies or communications between the threads and each thread is executed and treated by the emulator independently.

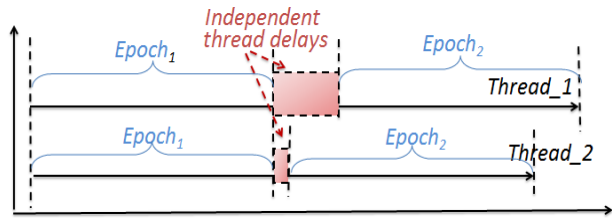
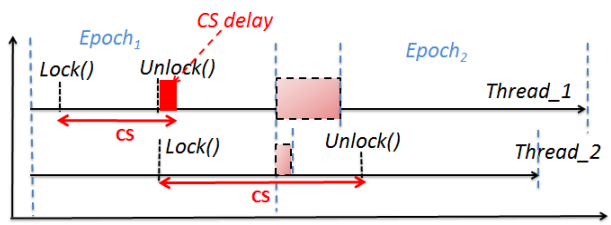
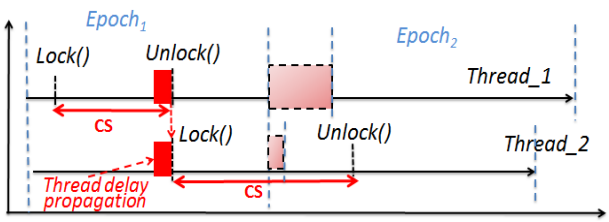


Figure 3: Emulating NVM latency for multithreaded application with two independent, asynchronous threads.

However, in a general case of multithreaded applications, naively injecting delays at fixed-size epochs is not sufficient because of inter-thread dependencies and communication. For example, when one thread holds a lock for executing in a critical section, the second thread competing for the same lock should wait for release of this lock (see Figure 4 (a)). If the accumulated delay in the critical section of the first thread is injected after it releases the lock then the resulting threads execution schedule would be as if two critical sections of these threads were time-wise overlapped as shown in Figure 4 (a). In the correct latency emulation, the accumulated



(a) Multithreaded application with critical sections and independently injected delays



(b) Emulating execution of multithreaded application (with synchronization primitives) in NVM-based systems

Figure 4: Emulating NVM latency for multithreaded applications with synchronization primitives.

delay by the first thread (during its execution in the critical section) should be injected by the first thread at the end of critical section before releasing the lock, and in such a way to be propagated into the execution of the second thread (as a delay for getting the lock) as shown in Figure 4 (b). To capture dependencies, we extend our epoch model to close epochs and inject proper delays before any events that result in inter-thread communication, such as lock releases and condition-variable notifications.

Therefore, for modeling NVM latency during execution of multithreaded applications, the emulator must close the current epoch and open a new one, when the thread enters and/or exits a critical section so the *LDM_STALL* cycles generated in the critical section are correctly injected as delays before releasing or acquiring the lock. As a result, this delay will be propagated to other threads which are waiting to access the critical section. Figure 4 (b) presents this scenario.

Synchronization primitives are tracked by the emulator, which will cause closing the current epoch with injected delay and opening a new epoch. However, very frequent synchronization primitives and corresponding delay calculations by the emulator may introduce a high overhead and lead to emulation inaccuracy. Therefore, besides the maximum static (*maximum*) epoch duration, we introduce a *minimum* epoch duration which determines the amount of time a thread must execute before a new epoch can be open. This allows minimizing the emulator overhead without sacrificing its accuracy.

3. QUARTZ IMPLEMENTATION

In this section, we describe the implementation of Quartz, which is our platform for emulating persistent memory. For ease of exposition, we start by describing how Quartz implements the performance model of persistent memory presented in Section 2. Then, we cover extensions to the basic model and implementation for emulating the application execution on future systems with two types of memory: fast, regular volatile DRAM and slower persistent memory.

3.1 Emulating Persistent Memory

We implemented Quartz in Linux as a pair of a simple kernel module and a user-mode library that provides the core functionality for emulating persistent memory. Our current prototype runs on the three latest Intel processor architectures: Sandy Bridge, Ivy Bridge, and Haswell.

The kernel module serves the library in two ways. First, it lets the library throttle DRAM bandwidth in a per channel basis by programming the thermal control registers `THRT_PWR_DIMM_[0:2]` found in the integrated memory controller of modern Intel Xeon processors [6]. These registers are available through the PCI configuration space and require privileged access. We confirm that the throttling degree is linear in the space of the register size (12 bits) as we show later in the evaluation section. We estimate the maximum bandwidth possible for each register value by measuring the time to stream through a large memory region using x86 streaming instructions (SSE). To effectively saturate memory bandwidth, we fork multiple threads each of which uses streaming instructions to access a part of the region. The helper program saves these values for later use by the user-mode library.

Second, the kernel module programs the hardware performance monitoring counters (PMC) to count the performance events shown in Table 1, which are later used by the library to derive stall cycles at runtime.[3] It also enables direct access to the counters from user mode via `rdpmc` instructions to reduce the overhead for accessing counters. We find this method to have a lower overhead compared to accessing counters using frameworks like `perf` [11] and `PAPI` [9] that virtualize performance counters and require trapping into the kernel on each access.

Sandy	$L2_{stalls}$	CYCLE_ACTIVITY:STALLS_L2_PENDING
	$L3_{hit}$	MEM_LOAD_UOPS_RETIRED:L3_HIT
	$L3_{miss}$	MEM_LOAD_UOPS_MISC_RETIRED:LLC_MISS
Ivy	$L2_{stalls}$	CYCLE_ACTIVITY:STALLS_L2_PENDING
	$L3_{hit}$	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_NONE
	$L3^{local}_{miss}$	MEM_LOAD_UOPS_LLC_MISS_RETIRED:LOCAL_DRAM
	$L3^{remote}_{miss}$	MEM_LOAD_UOPS_LLC_MISS_RETIRED:REMOTE_DRAM
Haswell	$L2_{stalls}$	CYCLE_ACTIVITY:STALLS_L2_PENDING
	$L3_{hit}$	MEM_LOAD_UOPS_L3_HIT_RETIRED:XSNP_NONE
	$L3^{local}_{miss}$	MEM_LOAD_UOPS_L3_MISS_RETIRED:LOCAL_DRAM
	$L3^{remote}_{miss}$	MEM_LOAD_UOPS_L3_MISS_RETIRED:REMOTE_DRAM

Table 1: Performance events per processor family.

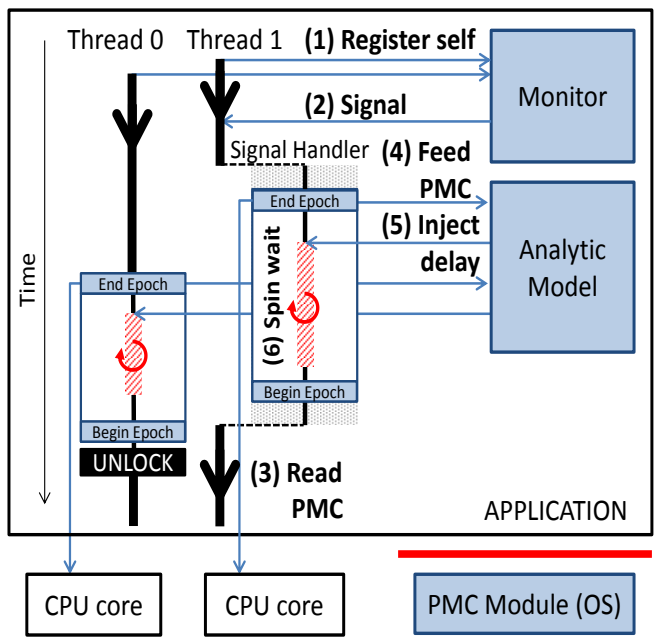


Figure 5: Interactions between applications threads and Quartz modules (solid blue color).

The user-mode library provides the core functionality for emulating PM. Upon loading, the library performs necessary initialization. First, it throttles DRAM bandwidth to the target PM bandwidth via the kernel module. Then, it forks a *monitor* thread that monitors application threads. The monitor learns about new application threads by interposing on thread creation events (`pthread_create`) so that new threads callback into the library and register themselves with the monitor (Figure 5, step 1). We implement function interposition by leveraging the fact that system library functions are usually defined as weak symbols and define a new function with the same name and signature that intercepts the original function call. We ensure our library can intercept such calls by loading the library before any other library using the LD_PRELOAD environment variable.

The library creates epochs in two ways. First, the monitor thread periodically wakes up and sends POSIX signals to interrupt each

[3]Note, that the performance events are the same on Ivy Bridge and Haswell with a small change in the event names: from "LLC" to "L3".

application thread whose current epoch time length exceeds a configurable *maximum* epoch length size (Figure 5, step 2). When a thread receives an interrupt signal, it ends the current epoch, reads the performance counters of Table 1 directly using `rdpmc` instructions (Figure 5, step 3), and then feeds the performance counter values into the analytic model (Figure 5, step 4). The model uses three performance counters to derive the stall cycles as defined in [2]:

$$LDM\_STALL = L2_{stalls} \times \frac{W \times L3_{miss}}{L3_{hit} + W \times L3_{miss}} \quad (3)$$

where W is the ratio of DRAM to L3 cache latency. *LDM_STALL* value is used in Eq 2 to estimate the injected delay as described earlier in section 2.2. It then injects the delay by spinning for the required time, and begins a new epoch (Figure 5, steps 5,6). We implement spinning through a software spin loop that uses the x86 `rdtscp` instruction to read the processor timestamp counter and which spins until the counter reaches the intended delay [36, 30]. The monitor thread estimates each thread's current epoch length by comparing the current monotonic timestamp (`clock_gettime`) to the timestamp that each thread records when it began its current epoch. Currently, the monitor thread wakes up at configurable fixed time intervals. This keeps the implementation simple but wake-up events and thread epoch completion times may slightly drift apart. We did not find this to dramatically impact accuracy.

A challenge with using signals is that signals may interrupt system calls and potentially break applications that do not properly handle system call errors due to interruptions. To mitigate this problem, applications that do not properly handle system-call interruptions can be extended to simply retry system calls.

The library also creates epochs at events that result in inter-thread dependencies. Our current prototype detects POSIX lock-release events by interposing on `pthread_mutex_unlock` but could be easily extended to interpose on other similar functions. The interposition code intercepts the original function call, ends the current epoch, injects the necessary delay, and creates a new epoch before redirecting the call to the actual function. We find, however, that frequent inter-thread dependencies, such as short contented critical sections, may result in many small epochs. When this happens the epoch creation overhead may be higher than the epoch itself, thus contributing to a very high emulation overhead. We mitigate this overhead by enforcing a configurable *minimum* epoch size. When a thread needs to begin a new epoch, it first checks whether the current epoch size exceeds the minimum size before ending and beginning a new epoch. As we show later in the evaluation section, this relaxation dramatically reduces overhead while not compromising accuracy.

Finally, the library provides applications with an extended API for working with persistent memory. First, it exports a `pmalloc/pfree` API that applications may use to allocate and free persistent memory. Second, it provides a `pflush` method that flushes and write-backs a cacheline out to memory (using the x86 `clflush` instruction), and then injects a configurable delay for emulating slower writes to persistent memory [36]. This approach to writes is different from regular volatile memory. In volatile memory, writes are posted, meaning the processor does not have to wait for the completion of a write before issuing the next one as long as there are available hardware buffers for holding outstanding writes. Thus, usually writes are not on the applications's critical path and do not contribute to processor stall cycles. In contrast, with persistent memory, writes are on the critical path because persistent-memory applications that correctly implement crash recovery require issuing and waiting for writes to complete in a specified order. So while the epoch-latency model is necessary for emulating slower NVM writes, it is not sufficient as it does not capture dependencies between writes on the critical path. The `pflush` method effectively captures such dependencies by pessimistically assuming that each write depends on the previous write to persistent memory, and emulates slow writes by simply stalling and waiting for a write to complete before continuing with the next write. Later, in Section 6, we discuss ideas on how to further improve on this approach and allow for parallelism in the case of independent writes.

3.2 Tuning the Emulation Overhead

Overall, the emulator's library initialization takes around 5.5 billion cycles, which translates into 2.5 seconds of wall clock time on a 2.2 GHz CPU with several NUMA nodes on the system. Registering an application thread takes 300,000 cycles (therefore, spawning an additional thread requires 10 microseconds on a 2.2 GHz CPU).

The epoch calculation takes on average 4000 cycles of overhead, where roughly half of it is imposed by the CPU performance counters reading. We make use of `rdpmc` instructions to directly read the CPU counters in user space in order to minimize overhead. Quartz keeps track of accumulated overhead that is caused by the epoch creation calls. The emulator tries to amortize this overhead by decreasing the injected delay for emulation of NVM latency. In case a calculated delay at the end of the current epoch is smaller than the overall accumulated overhead, this overhead is carried over to the upcoming epochs.

With a proper epoch duration configuration, the emulator is able to amortize all the epoch processing overhead. For memory bound applications, a smaller epoch size is beneficial. Quartz is augmented with specially designed statistics to provide useful feedback to the user: this statistics reports whether the emulator overhead was amortized entirely or not, and it indicates whether adjusting the epoch size may improve emulation accuracy for a particular application. Quartz offers a few additional knobs in its configuration for measuring the emulation overhead. These knobs are useful for tuning the emulator parameters, such as selecting a right epoch size for emulating a given workload. In particular, a user can run the emulator in a special mode with a "switched-off" delay injection, while preserving the calls for epoch creation and all the related delay calculations. Then the user can assess the emulator overhead for epoch creation by comparing the application completion time with "no emulation" versus the application completion time with the emulator in a "switched-off" delay injection mode. For most experiments reported in this paper the emulator overhead (related to epoch creation) was less than 4%.

Our choice of accessing performance counters using `rdpmc` instructions is mainly driven by a significantly lower overhead compared to standard frameworks such as `perf` [11] and PAPI [9]. These frameworks are attractive because they offer portable interfaces for accessing counters. However, they come with a high overhead price. For example, accessing all the required counters using PAPI tool will incur 30,000 cycles, which is about 8 times higher overhead compared to using `rdpmc` instructions. This would increase the epoch processing overhead to 30,000 cycles, and would make it very difficult to amortize this overhead, and therefore, would lead to a high emulation inaccuracy.

3.3 Emulating Two Memory Types: Volatile Memory and Persistent Memory

Emulating persistent memory provides us with an important framework for understanding its performance implications to future applications. However, future system architectures may also choose to provide DRAM volatile memory as a faster type of memory in addition to NVM persistent memory [28]. This opens up several questions about system software design with NVMs. Shall we build systems with two types of memory: DRAM (fast, small) and NVM (slow, large), i.e., where DRAM is not simply a caching layer for NVM? Given these two types of available memory, how

shall we design new applications or redesign the old ones to benefit from this memory arrangement and decide on the efficient data placement? Thus, many design and data placement decision will depend on the performance characteristics (latency and bandwidth) of future NVMs, and therefore, there is a need to extend the proposed performance emulator to mimic performance characteristics of both DRAM and NVM technologies.

In this section, we introduce Quartz' extension for emulating such system with two types of memory. Figure 6 presents the emerging future architecture, which we would like to emulate. Each socket (CPU) in the target system is directly attached to DRAM and NVM modules.

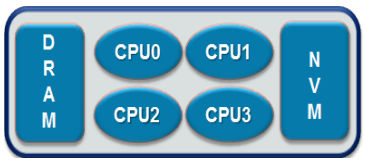


Figure 6: A future system with co-existed two types of memory: DRAM and NVM.

To implement the system configuration shown in Figure 6, we utilize a multi-socket server (two-socket servers are rather a new norm in the enterprise environments) and enhance Quartz design with a Virtual Topology feature as shown in Figure 7. At a first step, the emulator partitions CPUs (sockets) into sibling sets where each set contains two sibling CPUs. This enables the use of unmodified local DRAM latency while offering emulated (higher) access latency to virtual NVM.

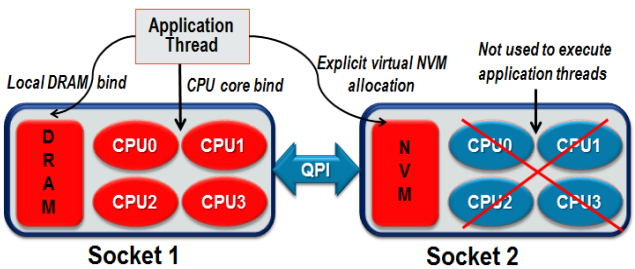


Figure 7: Quartz implementation that utilizes a two-socket server for emulating the co-existence of DRAM and NVM.

Next, all application threads are bound to the first CPU of each sibling set and respective local DRAM of the first CPU: for allocating regular volatile memory using the standard `malloc/free` API. Then the emulator maps virtual NVM to DRAM of the sibling CPU socket, referred to as remote DRAM. The emulator provides a `pmalloc/pfree` API that application threads have to use for allocating memory from virtual NVM. We utilize the underlying NUMA allocator methods (i.e., `numa_alloc_onnode`) to allocate memory from remote DRAM associated with the emulated NVM.

When designing new applications and algorithms that utilize both types of memory (DRAM and NVM), the application programmers can use the `malloc/free` API to aimingly allocate frequently accessed data structures in volatile memory, and use respectively the `pmalloc/pfree` API for mapping and storing larger (less-frequently accessed) data in NVM.

While assigning each memory type to separate sockets reduces the number of available cores in the system (as remote DRAM cores are not used for computation), it lets us leverage hardware performance counters to estimate memory references issued to volatile memory (i.e., local DRAM) and non-volatile memory (i.e., remote DRAM), as we explain next.

In order for the emulator to be able to selectively increase the access latency to NVM (emulated by remote DRAM) while leaving local memory accesses unchanged, first of all, we should be able to split the measured processor stall cycles into two portions:

- stall cycles that are due to *local memory* accesses, and
- stall cycles that are incurred by accessing *remote DRAM*.

Once we estimate stall cycles that are due to remote memory accesses, we could apply our model presented in Section 2.2 for emulating NVM latency.

To our rescue, the modern Intel processors (e.g., Ivy Bridge, Haswell) support hardware performance counters that report whether LLC (*Last Level Cache*) misses are served from local or remote DRAM. To calculate an additional delay which the emulator needs to inject at the epochs boundaries, we use the following notations:

- $DRAM_{lat}^{loc}$ - the average *local* DRAM access latency (in *ns*).
- $DRAM_{lat}^{rem}$ - the average *remote* DRAM access latency (in *ns*).
- M_i^{loc} - the total number of memory references going to *local* DRAM in epoch *i*.
- M_i^{rem} - the total number of memory references going to *remote* DRAM in epoch *i*.
- LDM_STALL_i - the *total* number of processor stall cycles caused by serving all memory requests in epoch *i*.

In order to correctly split the measured (total) stall cycles into local and remote memory stall cycles, we also need to take into account the measured access latencies to local and remote DRAM. Here is a simple example, explaining the intuition and the equations, which we present below. Let 3000 ns be reported as a total stall time in the latest epoch, and let there be 10 local and 10 remote memory references reported by hardware counters in this epoch. Assume that local DRAM latency is 100 ns and remote DRAM hardware latency is 200 ns. Note that higher remote latency directly translates into a higher processor stall time for remote accesses. Therefore, it will be incorrect to split 3000 ns equally between 10 local and 10 remote accesses. We have to use measured latencies for local and remote memory accesses as additional weights in splitting the stall time: 3000 ns = 10*100 ns +10*200 ns, and therefore, 1000 ns will be a portion of a local stall time while 2000 ns will be attributed to a remote stall time. Following this reasoning, we can calculate the stall time due to remote memory accesses by using the following heuristic:

$$LDM\_STALL_i^{rem} = LDM\_STALL_i \times \frac{M_i^{rem} \cdot DRAM_{lat}^{rem}}{M_i^{loc} \cdot DRAM_{lat}^{loc} + M_i^{rem} \cdot DRAM_{lat}^{rem}}$$

Once we estimate $LDM\_STALL_i^{rem}$ for each epoch *i*, we can follow the memory model designed in Section 2.2 and compute the required software delays for emulating NVM latency.

Therefore, for implementing the system with two types of memory: DRAM and NVM, the designed model requires at most *four hardware performance counters* available in Ivy Bridge and Haswell processors.

4. PERFORMANCE EVALUATION STUDY

In this section, we evaluate the accuracy and the utility of the proposed approach by using a set of specially designed microbenchmarks and Big Data applications. In particular, we utilize a variety of specially tailored microbenchmark, such as *MemLat, Multi-Threaded*, and *MultiLat* benchmarks, designed for validating specific features of the emulator. We validate the emulator implementation on three latest Intel Xeon-based processor architectures: *Sandy Bridge, Ivy Bridge*, and *Haswell*. In some cases, we present the execution/validation results for all three hardware choices in

our testbed. However, to avoid, the presentation repetitiveness (due to similarity of these results), for most cases, we present the subset of performed experimental results.

4.1 Experimental Testbeds

Our emulator is implemented and evaluated on three different dual-socket systems representing latest Intel Xeon-based processor families:

- **Intel Xeon E5-2450 with Sandy Bridge processor** that supports 16 *two-way* hyper-threaded cores running at 2.1 GHz, with measured local and remote memory access latencies of 97 *ns* and 162 *ns* respectively.
- **Intel Xeon E5-2660 v2 with Ivy Bridge processor** that supports 20 *two-way* hyper-threaded cores running at 2.2 GHz, with measured local and remote memory access latencies of 87 *ns* and 176 *ns* respectively;
- **Intel Xeon CPU E5-2650 v3 with Haswell processor** that supports 20 *two-way* hyper-threaded cores running at 2.3 GHz, with measured local and remote memory access latencies of 120 *ns* and 175 *ns* respectively.

Table 2 presents the summary of measured accesses latencies to local and remote DRAM in different testbed servers used in our performance experiments:

Processor Family	*Min local*	*Aver local*	*Max local*	*Min remote*	*Aver remote*	*Max remote*
Sandy Bridge	97	97	98	158	163	165
Ivy Bridge	87	87	87	172	176	185
Haswell	120	120	120	174	175	175

Table 2: Measured Memory Access Latencies.

4.2 Validating Accuracy of Memory Bandwidth Emulation

As bandwidth emulation is solely based on hardware features, we are primarily interested in verifying that the memory bandwidth can be indeed controlled through the thermal control registers.

Figure 8 shows the memory bandwidth measured using *copy* kernel of STREAM benchmark [14] for varying thermal control register values. The measured memory bandwidth changes linearly as a function of specified register values, until the application's maximum attainable bandwidth is reached.

As DRAM memory bandwidth can be controlled *linearly* using thermal control registers, we conclude that the desired bandwidth for NVM can be realized with good accuracy.

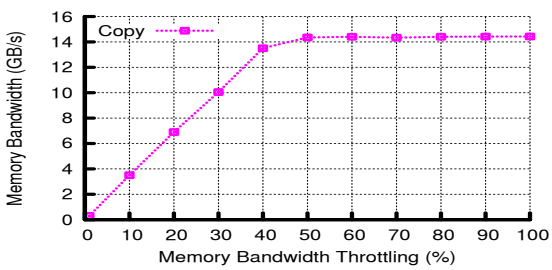


Figure 8: Relationship between memory throttling using thermal control registers and memory bandwidth of STREAM benchmark (*copy* kernel) on Sandy Bridge processor.

4.3 Approach for Validating Memory Latency Emulation

An additional interesting challenge in building the NVM performance emulator is solving the problem "how to validate the correctness and accuracy of the designed emulation platform"? Since the next-generation of NVMs are not currently available, it is a non-trivial task to assess the effectiveness of our approach and accuracy of performance models used in the emulator design.

In order to achieve "physically slower" memory (i.e., memory with higher access latency), we perform our experiments on a multi-socket machine as shown in Figure 9, and utilize NUMA (*Non-Unified Memory Access*) properties of these servers, i.e., the fact that processor accesses to remote DRAM exhibit higher latencies compared to local memory access time.

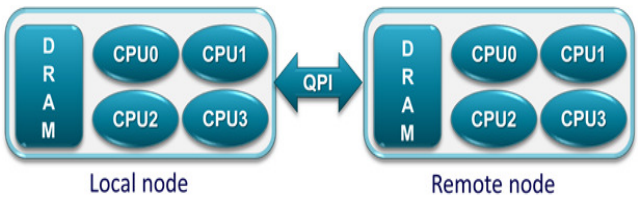
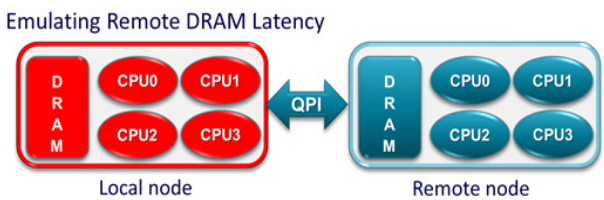


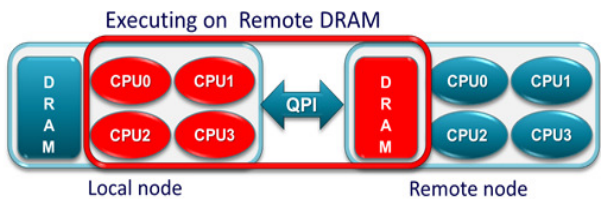
Figure 9: Two-socket servers used in our testbed.

Therefore, we utilize the following **two different configurations** in our performance experiments:

- **Conf_1** – where both a processor and allocated memory belong to the same socket. Then we use our emulator to enforce higher memory access latency and emulate slower NVM for executing benchmarks and test applications in this configuration as shown in Figure 10 (a), and in particular, we can emulate the remote DRAM access latency;
- **Conf_2** – where we use a processor from one socket and allocated remote memory from the other socket to run the same applications as shown in Figure 10 (b). We use the *numactl* tool [8] to bind the experiment's computation on the local socket and force the experiment to use memory from the remote socket: this way we can physically increase memory latency.



(a) Executing benchmarks and applications under Quartz and higher (**emulated**) memory latency in **Conf_1**.



(b) Executing benchmarks and applications with higher (**physical**) memory latency in **Conf_2**.

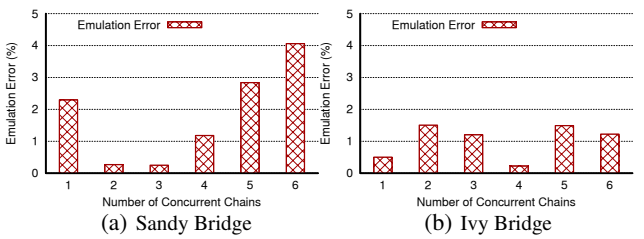
Figure 10: Validation testbed.

First, we run a set of latency-sensitive experiments on *Conf_1 with our emulator* which injects software created delays to mimic the latency of remote socket memory. Then for validation and comparison, we measure the application completion times when they are executed directly on *Conf_2 without the emulator*.

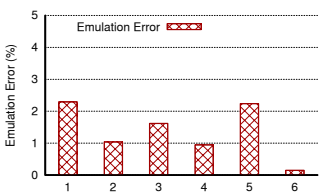
4.4 Validating Memory Model Implemented in Quartz

We designed a memory-latency bound pointer-chasing benchmark, called *MemLat*, with a configurable degree of memory access parallelism. The benchmark creates a pointer chain as an array of 64-bit integer elements. The contents of each element dictate which one is read next; and each element is read exactly once. We choose the array size to be much larger than the size of the last-level cache so that each element's memory access results in a cache miss that is guaranteed to be served from memory.

The *MemLat* microbenchmark is *memory-latency sensitive* because the next element to be accessed is determined only after the current access completes. The benchmark can also create *multiple* independent chains to define *different degrees of memory access parallelism* used for validation of our memory model that accounts for MLP (Memory Level Parallelism). During each iteration the microbenchmark accesses the current element of each chain before proceeding with the next element. This results in multiple parallel memory requests as element accesses from different chains are independent. To minimize memory accesses due to TLB misses, we configure the virtual memory subsystem to use 2 MB hugepages.



(a) Sandy Bridge (b) Ivy Bridge



(c) Haswell

Figure 11: Validation experiments with *MemLat* benchmark.

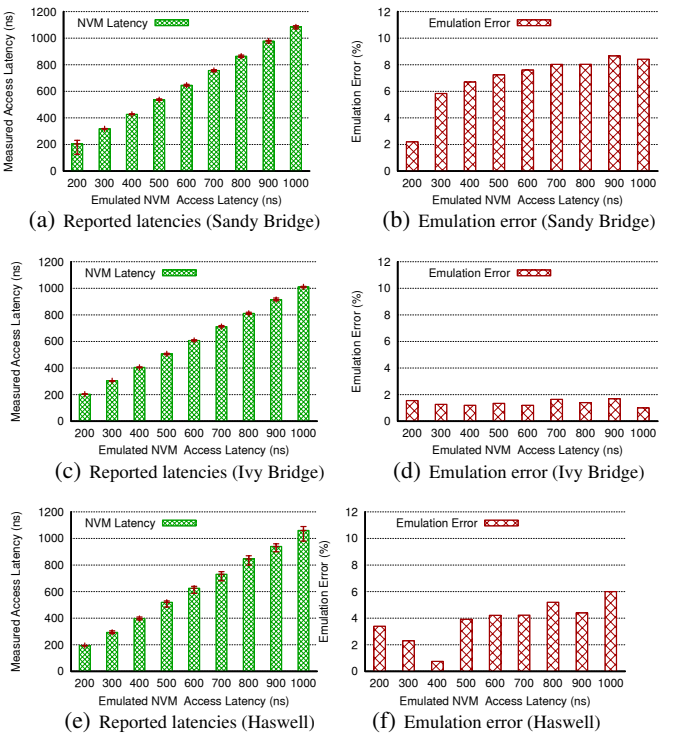
Figures 11 (a)-(c) report the outcome of our validation experiments performed for three latest Intel Xeon-based processors: Sandy Bridge, Ivy Bridge, and Haswell. We executed *MemLat* in *Conf_1* (on local memory) under the emulator which was configured to emulate the remote memory access latency. Then we compared these measurements against the actual execution times of *MemLat* by running it in *Conf_2* (directly on remote memory). The X-axis reflects the number of concurrent pointers chains configured for the *MemLat* benchmark execution, and which defines the number of independent memory accesses issued by microbenchmark at each iteration, i.e., 1, 2, 3, 4, 5, and 8, and Y-axis shows the emulation errors (%) [4].

Figures 11 (a)-(c) show that Quartz accurately emulates *MemLat* executions with different concurrency degree compared to the measured results of the same benchmark on the remote DRAM in *Conf_2*. The emulated and measured results are only 0.2%-4% apart from each other for all three processors: Sandy Bridge, Ivy Bridge, and Haswell.

[4] All the experiments are performed 20 times, and the measurement results are averaged. This comment applies to all results in this section. We performed the experiments for different maximum epoch sizes, e.g., 1 ms, 10 ms, and 100 ms. The accuracy degrades with larger epoch size, e.g., 100 ms, while 1 ms and 10 ms epochs support a good accuracy. We use 10 ms epochs to minimize the overhead.

An interesting and useful feature of the implemented *MemLat* benchmark is that it can be used for measuring the memory access latency[5]. Therefore, we exploit this *MemLat* functionality in our experiments. When we set Quartz to emulate a specified memory latency Lat_{emul}, we could compare it against the measured memory latency Lat_{meas} reported by *MemLat*, and in such a way, we can compute the emulation error.

Figures 12 (a),(c),(e) present the results of executing *MemLat* with Quartz for the three processor families (Sandy Bridge, Ivy Bridge, and Haswell) while emulating a wide range of NVM latencies (shown in X axes): from 200 ns to 1000 ns. X-axis show the emulated memory latencies Lat_{emul}, while Y-axes show the reported by *MemLat* measured memory access latencies Lat_{meas}. The red error bars show the variation in measured times across 20 trials.



(a) Reported latencies (Sandy Bridge) (b) Emulation error (Sandy Bridge)
(c) Reported latencies (Ivy Bridge) (d) Emulation error (Ivy Bridge)
(e) Reported latencies (Haswell) (f) Emulation error (Haswell)

Figure 12: Reported latencies by *MemLat* under Quartz emulator with different target (emulated) NVM latencies.

Figures 12 (b),(d),(f) show the emulation errors, i.e., the difference between target emulated NVM latencies and the measured ones. Quartz demonstrates a good accuracy across all the experiments: the errors are:

- less than 9% on Sandy Bridge processor family,
- less than 2% on the Ivy Bridge processors, and
- less than 6% on Haswell processors.

The difference in performance results is mainly due to a difference in hardware performance counters available (for accounting the *stall cycles)* in these processor families[6].

4.5 Validating Quartz Implementation for Multi-Threaded Case

The main challenge in correctly emulating multi-threaded applications is propagating delays between the threads with inter-

[5] *Memory Latency Checker* tool from Intel exploits a similar idea [4].

[6] The counters available in earlier Intel Sandy Bridge processor family are less reliable as many blogs' messages and discussion groups reflect.

dependencies (as shown in Fig. 4). Quartz introduces configurable *minimum* epochs for propagating these delays at thread synchronization points in addition to a static (*maximum*) epoch mechanism.

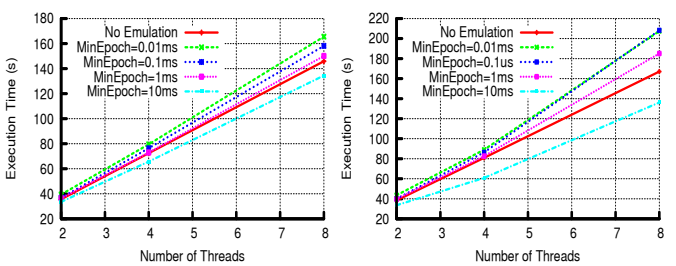
To validate and analyze the proposed solution, we designed a special *Multi-Threaded* benchmark, which can be configured with the following parameters:

- *N* - to spawn *N* threads;
- *K* - each thread executes *K* critical sections;
- *cs_dur* - with computations inside the critical sections defined by the *cs_dur* number of pointer chasing iterations of the *MemLat* benchmark;
- *out_dur* - with computations outside (between) the critical sections defined by the *out_dur* number of pointer chasing iterations of the *MemLat* benchmark.

We measure *Multi-Threaded* benchmark execution times in *Conf_1* under our emulator, which emulates on local DRAM the access latency of remote DRAM. Then we compare these measurements against the actual execution times of the *Multi-Threaded* benchmark (without the emulator) in *Conf_2* (i.e., by directly executing it on remote DRAM).

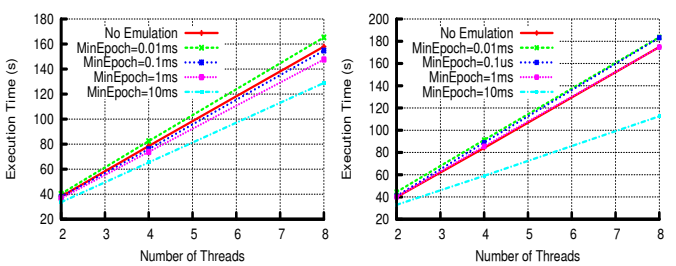
To analyze the benefit and overhead of the "minimum" epoch creation, we run experiments with 2, 4, and 8 threads, with each thread entering 1 million critical sections, where it executes 100 pointer chasing iterations.

Figures 13 (a)-(d) summarize these results on Sandy Bridge and Ivy Bridge processor families for two cases. Figures 13 (a),(c) represent an extreme case when each thread computes in the **critical sections only** and no computation is done outside the critical sections (abbreviated as *cs only*). While Figures 13 (b),(d) represent a more traditional scenario, where a thread performs **computations inside and outside of a critical section**: we applied parameters that define an equal amount of computations inside and outside of a critical section (abbreviated as: "with compute").



(a) Sandy Bridge (cs only)

(b) Sandy Bridge (with compute)



(c) Ivy Bridge (cs only)

(d) Ivy Bridge (with compute)

Figure 13: Validating Quartz emulation accuracy for *Multi-Threaded* benchmark (on Sandy Bridge and Ivy Bridge processors).

Each figure has 5 lines: the red solid line which shows the actual benchmark execution times (no emulation), and four lines that correspond to *Multi-Threaded* benchmark executions with Quartz emulator configured with different minimum epoch size of 0.01 ms, 0.1 ms, 1 ms, and 10 ms. Note, that since we use the maximum (static) epoch size of 10 ms across our experiments, the case with minimum and maximum epoch sizes of 10 ms (light blue line) shows what would happen if each thread injects its own (independent) delay, i.e., the delays due to inter-dependencies between the threads are not propagated. Indeed, in this case, the emulation does suffer a high inaccuracy, which gets worse with the increased number of threads (up to 34% for Ivy Bridge).

These results justify our design and implementation with intercepting the synchronization primitives and injecting the additional delays using the *minimum* epochs' mechanism. The remaining three lines show a good accuracy of Quartz (<3% error) compared to actual measurements and the insignificant implementation overhead.

4.6 Validating the Emulator Implementation with DRAM+NVM Configuration

In Section 3.3, we described our approach for emulating systems with two types of memory: DRAM (fast, small) and NVM (slow, large) by utilizing two-socket servers. Under this approach the CPU from one socket is used to support application processing on data allocated in local DRAM (with unchanged access latency) and remote DRAM (which is used for emulating NVM with a higher access latency).

The main challenge in correctly emulating performance of an application which utilizes both types of memory (DRAM and NVM) is the ability of Quartz to accurately split the accumulated processor stall cycles into two parts: 1) stall cycles which are accumulated as a result of local DRAM accesses and 2) stall cycles which are due to memory accesses to remote DRAM. If emulator has an estimate of processor stalls due to remote DRAM accesses then it can compute a required additional delay for injection into the application execution to emulate a specified (higher) NVM access latency.

To validate and analyze the proposed solution, we designed a special *MultiLat* benchmark, which is a tailored extension of the *MemLat* benchmark described in Section 4.4. The original *MemLat* benchmark creates a pointer chain as an array of 64-bit integer elements. The contents of each element dictate which one is read next; and each element is read exactly once. The new *MultiLat* benchmark creates a pointer chain over two different arrays (with the specified array sizes), where one array is placed in DRAM while the second one is mapped into NVM. In addition, *MultiLat* let the user specify the access pattern in the benchmark's sequence of memory accesses (i.e., in a pointer chain). This access pattern is iterative (repetitive) and defines the number of DRAM accesses in the sequence followed by the number of NVM accesses before the pattern is repeated. For example, we can define an access pattern where the benchmark after each 200 DRAM accesses performs 100 NVM accesses, etc.

Intuitively, if we implemented the emulator correctly then the completion time of such a benchmark is defined by the sum of the number of elements in each array multiplied by the corresponding memory access time. Therefore, the benchmark completion time should be the same (or very close) for different access patterns. For example, let we execute *MultiLat* benchmark configured with the following parameters:

- Num^{DRAM} – the number of elements in the array residing in DRAM;
- Num^{NVM} – the number of elements in the second array placed in NVM;
- $DRAM_{lat}$ – the measured DRAM access time;
- NVM_{lat} – the specified (emulated) NVM access latency.

Then for different memory access patterns (that we can define using

the design of this benchmark) its completion time CT should be:

$$CT = Num^{DRAM} \cdot DRAM_{lat} + Num^{NVM} \cdot NVM_{lat}$$

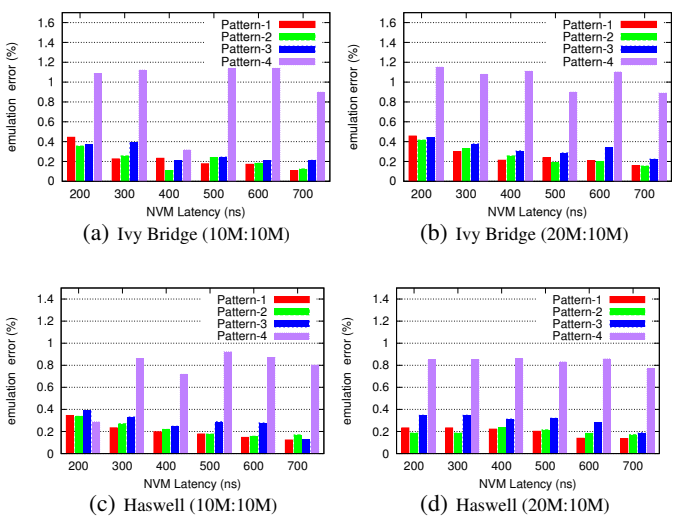
We have executed *MultiLat* benchmark with two different configurations, where

- Configuration **10M:10M** denotes the *MultiLat* benchmark with $Num^{DRAM} = 10M$ and $Num^{NVM} = 10M$;
- Configuration **20M:10M** denotes the *MultiLat* benchmark with $Num^{DRAM} = 20M$ and $Num^{NVM} = 10M$.

In addition, we executed these two *MultiLat* configurations with **four different access patterns**:

1. *Pattern-1* defines a recursive pattern where 200,000 DRAM accesses are followed by 100,000 NVM accesses, etc.
2. *Pattern-2* defines a recursive pattern where 20,000 DRAM accesses are followed by 10,000 NVM accesses, etc.
3. *Pattern-3* defines a recursive pattern where 2000 DRAM accesses are followed by 1000 NVM accesses, etc.
4. *Pattern-4* defines a recursive pattern where 200 DRAM accesses are followed by 100 NVM accesses, etc.

Figures 14 show the average emulation errors (Y-axis) observed in these experiments with *MultiLat* for a set of emulated NVM latencies: 200 ns, 300 ns, 400 ns, 500 ns, 600 ns, and 700 ns (as shown in X-axis). We performed each experiment 10 times (for two different configuration and four different access pattern). The results are shown for Ivy Bridge and Haswell Xeon-based processor families.



(a) Ivy Bridge (10M:10M) (b) Ivy Bridge (20M:10M)
(c) Haswell (10M:10M) (d) Haswell (20M:10M)

Figure 14: Emulation error for *MultiLat* under Quartz emulator with different emulated NVM latencies and DRAM + NVM access patterns on the two-socket processors: Ivy Bridge and Haswell.

The average measured emulation errors are below 1.2% in these experiments for all access patterns and two *MultiLat* benchmark configurations.

4.7 Case Study with Real Applications

We performed a validation and sensitivity study with Quartz and two Big Data applications (production quality) such as *MassTree* [26] (that represents a fast Key-Value store) and a parallel PageRank algorithm from Yahoo [23].

In the experiments with *PageRank*, we used a graph with 4,847,571 vertices and 68,993,773 edges. The computation converges after 64 iterations with less than 9.563e-08 error.

Validation Results.

We performed validation experiments with both *MassTree* and *PageRank* by running them on *Conf_1 with our emulator* which injects software created delays to mimic the latency of remote socket memory. Then for validation and comparison, we measure the applications' performance when they are executed directly on *Conf_2 without the emulator*.

The implementation of *MassTree* allowed us to execute this application with different number of threads. We perform the experiments with 1,2,4, and 8 threads[7]. Figure 15 shows two bars that represent the measured errors for *put/s* and *get/s* operations.

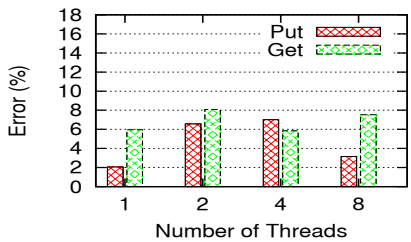


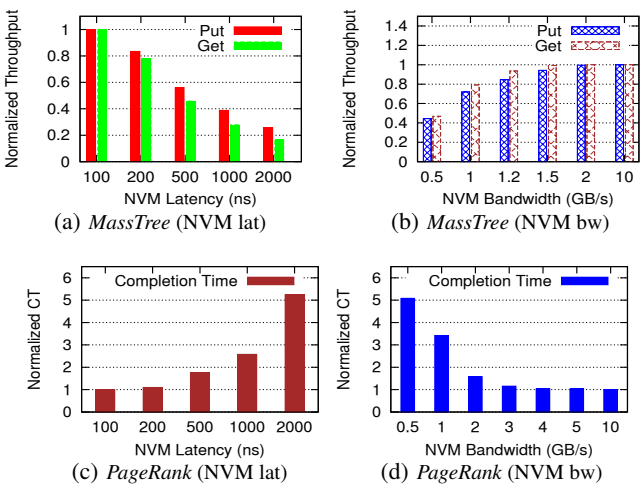
Figure 15: Validation errors for multithreaded executions of *MassTree* on Sandy Bridge processor family.

The emulation errors on Sandy Bridge vary between **2-8%** which shows a good accuracy of our emulator.

The *PageRank* implementation is single threaded (therefore, we only have validation numbers for this configuration). The difference (error) between emulated and measured completion times of *PageRank* on Sandy Bridge processor family is **2.9%**

NVM Latency and Bandwidth Sensitivity Results.

For sensitivity study we performed experiments, where we varied NVM latency and NVM bandwidth to analyze their impact on performance of selected applications.



(a) *MassTree* (NVM lat) (b) *MassTree* (NVM bw)
(c) *PageRank* (NVM lat) (d) *PageRank* (NVM bw)

Figure 16: *PageRank* and *MassTree* sensitivity to NVM latency and NVM bandwidth (Sandy Bridge processor family).

The results shown in Figures 16 (a),(c) present a non-linear performance degradation with increased NVM latency. If NVM latency is 200 ns (i.e., 2 x local DRAM latency) then *MassTree* throughput falls by 15%, while the completion time (CT) of *PageRank* is practically unchanged. However, as NVM latency reaches

[7] Each experiment is repeated 10 times and results are averaged. In order to eliminate a possible caching effect on the application performance, we invalidate caches between the runs.

2 μs *MassTree* throughput falls by almost 5 times, and the completion time of *PageRank* increases by more than 5 times (note, a non-linear scale of X-axis).

Figures 16 (b),(d) show that both applications, *PageRank* and *MassTree*, are not very sensitivity to NVM bandwidth. For *PageRank* application only when NVM bandwidth is less than 3 GB/s it starts impacting the *PageRank* completion time. For *MassTree* performance we can see that its throughput gets impacted only when NVM bandwidth is less than 1.5 GB/s.

Both applications are more sensitive to memory access latency than to memory bandwidth.

5. RELATED WORK

Several previous projects attempted to emulate performance of NVM using DRAM. Dulloor et al. [21] describe an emulation platform that requires special hardware and firmware. Similar to our approach, they inject delays derived using a simple stall model. However, they rely on special hardware hooks to monitor the amount of time a core is not committing instructions. Instead, we base our model on performance counters commonly available in commodity processors.

Pelley et al. [30] utilize *offline analysis* by applying PIN binary instrumentation tool to estimate the average number of cache misses per program regions. They use these estimates to introduce additional delays during actual runs on bare metal. Instead, we focus on an *online model* that does not require the extra step of offline analysis. There are two shortcomings to this approach, first it requires an offline training of the application. Second they do not take into consideration the memory level parallelism in the hardware.

Volos et al. [36, 35] emulate the NVM write-latency by injecting a software created delay, whenever a programmer explicitly flushes a cache line out of the processor. The memory latency model presented in our paper extends the earlier model by injecting delays to account for slow NVM reads.

In [24], the authors have proposed to use DRAM bandwidth throttling for impacting the application perceived latency (as a result of created resource contention). This method can only be applied if the application's bandwidth requirement is higher than the throttled bandwidth. Moreover, there is a difficulty with achieving an accurate latency "slowdown" using this method, and it does not work for latency-bound applications whose bandwidth requirements are very small. In contrast, our approach decouples latency and bandwidth emulations.

There is a body of works that utilize performance counters for analyzing application memory performance. Green governor model [22] monitors the last level cache (LLC) misses and memory stall cycles. They multiply average memory latency by LLC misses to get an estimated time spent in memory. However, their model ignores memory-level parallelism, which might lead to an over estimate of the actual memory time. Recent work [34] proposes a memory model based on *miss handling status register (MSHR)* introduced in AMD processors. However, these performance counters are not readily available in other platforms (e.g., Intel).

6. DISCUSSION: CHALLENGES AND OPPORTUNITIES

Challenges.

There are quite a few obstacles and challenges in accurately measuring the system hardware memory access latencies as one can see in the related user discussion at the Intel forum [5]. In our work, we faced similar challenges. For example, Quartz's interface allows a user to set a target NVM latency in nanoseconds. At the same time, performance counter measurements are provided in cycles, e.g., stall cycles. Therefore, we need to translate durations measured in cycles to durations in nanoseconds and vice versa. Typically, the specified processor frequency can be used for this translation. However, for energy efficiency many processors are applying DVFS (Dynamic Voltage and Frequency Scaling)[8] during workload processing depending on the system utilization. Therefore, to preserve a fixed relationship between cycles and time we disable DVFS feature.

Another issue is that a memory workload dynamically affects measured memory latency, i.e., measured "loaded" memory latency varies depending on the memory system utilization. We believe that our current model works for this scenario as well, but we plan to explore this issue in more detail and refine Quartz implementation and its memory model (if needed) to take this variance into account.

Finally, there are some remaining open questions related to the accuracy of the analytic memory model for workloads where memory accesses overlap with computation. When some degree of overlapping is available, memory accesses may not stall the processor completely for the entire trip to DRAM, but could still stall significantly with longer NVM latencies. Since the analytic model injects delay proportional to the cycles that the processor is stalled when accessing DRAM, it may inject less delay than needed to emulate the longer NVM latency.

Opportunities.

We are working on extending Quartz to emulate the range of NVM "store" latencies by exploiting a set of new operations x86 instructions specified by Intel [3, 15] for correctly handling writes to persistent memory, such as `clflushopt` and `pcommit`. The new `clflushopt` instruction writebacks and flushes a cacheline similarly to `clflush`, but unlike `clflush` the processor does not stall wait for the flush to complete with respect to NVM before continuing execution. Instead, a processor stall-waits for all previous flushes to complete when it reaches a `pcommit` instruction. This approach to write-flushes enables applications to express multiple independent writes that can proceed in parallel, such as when initializing multiple fields of a persistent object.

Our current approach to emulating writes through `pflush` (Section 3.1), where we stall-wait for a write to complete with respect to NVM before continuing with the next write, would pessimistically serialize such independent writes. To model multiple independent writes, we are planning to model `pcommit`. A model based on `pcommit` would accumulate delays when issuing flushes, and inject the accumulated delay at the end of the barrier rather than inject a delay at each flush. This enables the model to capture write parallelism by discounting from the accumulated delay flushes that are expected to complete by the time the program reaches the pcommit barrier.

We see an additional opportunity in integrating Quartz with QEMU (an open source machine emulator and virtualizer). When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g., an ARM-based system) on a different machine (e.g., Intel-based system). By using dynamic translation, QEMU achieves a low overhead and good performance. This integration will enhance Quartz' usability for studying NVMs with different performance characteristics on a variety of different systems.

7. CONCLUSION AND FUTURE WORK

We presented Quartz a performance emulator for persistent memory software. We show that our novel epoch-based latency

[8] DVFS is a power management technique, where the processor frequency can be increased or decreased dynamically.

emulation approach can leverage hardware performance counters found in modern processors to model average application perceived latency with low overhead and good accuracy.

Currently, we have got an access to a unique HP custom-built system that can change (increase) the memory access latency in hardware. Our preliminary experiments with Quartz versus hardware-based memory emulation do show good accuracy of Quartz (within 12% error for tests with a subset of our benchmarks and experiments with MassTree application as well as Graph500 benchmark [1] (its reference implementation). We plan to prepare an extended version of this paper with additional validation experiments as HP Labs Technical Report published later during this year. We also consider a possibility to make the code of the emulator available as an open source.

We believe that relying on commodity hardware will enable wider adoption of our platform over approaches that require special hardware. We hope the research community will find our emulator a useful tool for studying the performance of persistent memory software. Moving forward, we are also looking at supporting more events that cause inter-thread dependencies including context switches, inter-process communication, and other parallel programming constructs such as OpenMP primitives. Also, we are looking for ways to extend the applicability of our work. We are working on a few new applications that can demostrate how they can take advantage of a system that is equipped with both persistent memory and regular volatile DRAM-based main memory. We plan to perform and demonstrate sensitivity analysis which would enable system designers to study design trade-offs that arise from having two memory types with different speeds, such as deciding on data placement.

Acknowledgements

We would like to thank Prof. David Gleigh for his generosity in sharing the code of PageRank, a related dataset, and his kind help in the compilation process. Our warmest thanks to our colleagues from HP Labs RESS, Narayan Krishnan, Eric Wu, Annabelle Eseo, David Connell for their help in provisioning us with a variety of hardware testbeds. Special thanks to our HP colleagues Gary Gostin, Toshi Kani, and Sai Chalamalasetti for providing a unique system with hardware based latency emulation for our experiments and all useful explanations on its setup and usage.

8. REFERENCES

[1] Graph500. http://www.graph500.org/.

[2] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[3] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[4] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[5] Intel Memory Latency Checker v2.0 released. https://software.intel.com/en-us/forums/topic/517571.

[6] Intel xeon e5-product family. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[7] Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family.) http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[8] numactl - Control NUMA policy for processes or shared memory. http://linux.die.net/man/8/numactl/.

[9] PAPI: Performance application programming interface. http://icl.cs.utk.edu/papi/.

[10] PCMSIM: A simple PCM block device simulator for linux. https://code.google.com/p/pcmsim/.

[11] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[12] pmem.io: Persistent memory programming. http://pmem.io/.

[13] SNIA NVM programming technical work group. http://www.snia.org/forums/sssi/nvmp.

[14] STREAM benchmark: http://www.cs.virginia.edu/stream/.

[15] Why Intel Added the CLWB and PCOMMIT Instructions. http://danluu.com/clwb-pcommit/.

[16] With The Machine, HP May Have Invented a New Kind of Computer. http://www.businessweek.com/articles/2014-06-11/with-the-machine-hp-may-have-invented-a-new-kind-of-computer.

[17] K. Asanovic. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proc. of FAST*, 2014.

[18] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Proc. of MICRO'43*, 2010.

[19] 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*, ASPLOS XVI, 2011.

[20] 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 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[21] S. R. Dulloor, S. K. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems*, Apr. 2014.

[22] S. Eyerman and L. Eeckhout. A Counter Architecture for Online DVFS Profitability Estimation. *IEEE Transactions on Computers*, 59(11), 2010.

[23] D. Gleigh, L. Zhukov, and P. Berkhin. PageRank: A Linear System Approach. Technical report, YRL-2004-038, Yahoo! Research Labs, 2004.

[24] H. Hanson and K. Rajamani. What computer architects need to know about memory throttling. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA)*, 2010.

[25] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase-change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[26] Y. Mao, E. Kohler, and R. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *In Proc. of Eurosys*, 2012.

[27] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[28] K. Packard. The Machine Architecture. `http://keithp.com/blogs/the_machine_architecture`.

[29] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture(ISCA)*, 2014.

[30] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVM Era. *Proceedings of the VLDB Endowment*, Volume 7(2), 2013.

[31] M. K. Qureshi, J. K. Michele, Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-Based. main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[32] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA'09, 2009.

[33] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan. A framework for emulating non-volatile memory systems with different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (WIP)*, 2015.

[34] B. Su, J. L. Greathouse, J. Gu, M. Boyer, and Z. Wang. Implementing a Leading Loads Performance Predictor on Commodity Processors. In *Proc. of Usenix ATC*, 2014.

[35] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proc. of EuroSys*, 2014.

[36] 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*, ASPLOS XVI, 2011.