A Framework for Emulating Non-Volatile Memory Systems with Different Performance Characteristics

Dipanjan Sengupta[1,2], Qi Wang[1,3], Haris Volos[1], Ludmila Cherkasova[1], Jun Li[1], Guilherme Magalhaes[4], and Karsten Schwan[2]

[1]Hewlett-Packard Labs, [2]Georgia Institute of Technology, [3]The George Washington University, [4]Hewlett-Packard

[2]dsengupta6@gatech.edu, [3]interwq@gwu.edu, [1,4]{haris.volos, lucy.cherkasova, jun.li, guilherme.magalhaes}@hp.com, [2]karsten.schwan@cc.gatech.edu

ABSTRACT

Exponential increase of online data and a corresponding growth of data-centric applications (Big Data analytics) forces system architects to revisit assumptions and requirements of the future system design. New non-volatile memory (NVM) technologies, such as Phase-Change Memory (PCM) and HP Memristor offer significantly improved latency and power efficiency compared to flash and hard drives. Many future systems are expected to have both DRAM and NVM. This can radically change system and software design, and enable new style of Big Data processing applications. However, the commercial unavailability of new NVMs technologies and uncertainty of their performance characteristics make it difficult to assess new system software stacks and to study their performance impact on future workloads. To bridge this gap and encourage an early design phase, we are building a DRAM-based performance emulation platform[1] , called *NVMpro*, that leverages features available in commodity hardware, to emulate different latency and bandwidth characteristics of future NVM technologies. *NVMpro* enables an efficient and accurate 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.

Categories and Subject Descriptors: C.4 [Computer System Organization] Performance of Systems, D.2.6.[Software] Programming Environments.

General Terms: Measurement, Performance, Design.

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

1. INTRODUCTION

Emerging byte-addressable, non-volatile memory technologies such as phase-change-memory and memristors offer an alternative to disk for persistence and provide performance within the order of magnitude of DRAM. Forward-looking projects like Firebox [7] and HP's The Machine [6] envision future scale-out machines that have enormous amount of non-volatile memories (NVMs). 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) and NVM (slow)?
- 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 decision might depend on the *performance characteristics (latency and bandwidth)* of future NVMs. However, NVMs are not commercially available yet, and a few existing hardware prototypes [8, 9] 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).

In this work, we introduce a novel performance emulation platform, called *NVMpro*, that we are implementing on top of existing DRAM to emulate **different performance characteristics** of future NVM technologies. *NVMpro* utilizes several features available in commodity hardware to "slow down" DRAM and emulate a wide range of NVM latencies and bandwidth characteristics that can be used for performance evaluation of emerging byte-addressable NVMs. Thus, we **are not** after an accurate simulation of NVM functionality, but rather after emulating the NVM performance characteristics.

Since the next-generation 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, we perform our experiments on a multi-socket machine with different access latencies for local and remote DRAM. In our validation experiments, we analyze a set of specially designed memory-intensive applications and SPEC CPU2006 benchmarks. The completion times of the test applications in the emulation platform are on average within 5% of the measured ones on the remote memory configuration. The remainder of the paper presents our results in more detail.

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.

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 [9].

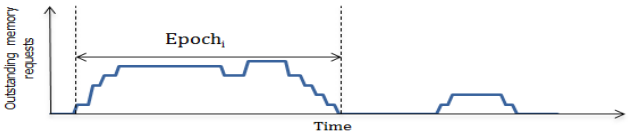
[1]This work was originated during Dipanjan Sengupta and Qi Wang internship at HP Labs.

ICPE'15, January 31-February 4, 2015, Austin, TX, USA.

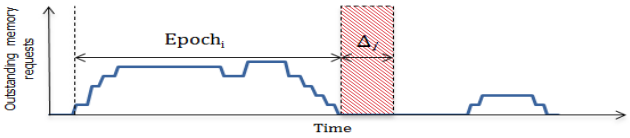
Specifically, we utilize thermal control registers found in the integrated memory controller of modern Intel Xeon processors [1] to programmatically throttle DRAM bandwidth in a per channel basis. The configuration registers we used are *THRT_PWR_DIMM_[0:2]*, and we use the *setpci* command to programmatically configure these thermal control registers.

Latency Model.

Unlike DRAM bandwidth, which can be programmatically controlled in modern processors as described above, modelling NVM latency is more challenging as commodity hardware does not provide a similar knob to physically control the DRAM latency. Therefore, we employ *a software-based solution* for emulating memory latency. As software 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 NVM latency. The **key idea** of the model is to dynamically inject software created delays to account for higher NVM latency of combined memory accesses. We avoid the overhead of instrumenting every memory access by adopting a coarse grain approach, in which we divide the application lifetime into time intervals called *epochs* and by inserting appropriate delays at the end of each epoch. Our emulator monitors and collects application's compute and memory characteristics for a given epoch using *hardware performance counters*, and at the end of each epoch it dynamically injects an appropriate amount of software delay in the application. The length of the epochs (and their frequency) is configurable. Figure 1 shows the memory access pattern of an application before and after the introduction of additional memory delay.



(a) Before the injection of software generated delay.



(b) After the injection of software generated delay

Figure 1: Emulation of NVM latency by injecting software delay at the end of each epoch.

The proposed model has two inter-related aspects: the logic behind the calculation of the additional delay for a given epoch, and the construction of the epochs, i.e., defining the size and frequency of these epochs.

For a *single threaded application* the epoch creation is as simple as creating the *fixed-size* intervals, but for multi-threaded applications it is more complex because of inter-thread dependencies and communications. In this work, we demonstrate our approach by considering a single threaded application model. (However, we have the model extention for the multi-threaded case, which is under performance evaluation.) In our model, we use the following denotations:

- NVM_{lat} - the average NVM access latency (in *ns*).
- $DRAM_{lat}$ - the average DRAM access latency (in *ns*).
- M_i - the total number of memory references going to the memory system in epoch i.
- LDM_STALL_i - the total number of processor stall cycles caused by serving memory requests in epoch i.
- Δ_i - software delay injected at the end of epoch i.

A very *simple* memory model for emulating the NVM latency is to count the total number of memory references made in a given epoch and multiply it by a difference in the average 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. Moreover, the hardware prefetching in modern processors can further reduce the memory references actually going and being served from DRAM. Therefore, we need to count 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

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

This *simple* model works correctly if we assume that all the memory references are issued serially to DRAM one after another. However, this assumption does not hold for most modern processors that support multiple memory requests to be issued and served in parallel. This feature is also known as *memory level parallelism (MLP)*. Figure 2 shows pictorially different memory reference processing patterns, that require injecting different delays for emulating a slower NVM latency.

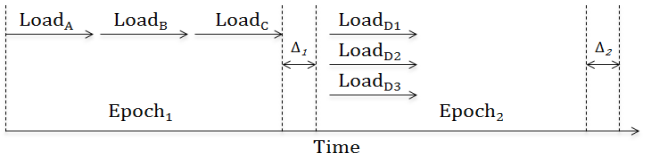


Figure 2: Impact of memory level parallelism on calculating the software delay injected at the end of each epoch.

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. However, in $Epoch_2$, this *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 *sequential* memory accesses for computing the additional delay in a given epoch i as follows:

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

Therefore, using only one hardware performance counter that measures LDM_STALL_i, the equation Eq. 2 computes the additional delay per epoch.

3. EVALUATION

In this section, we evaluate the accuracy of the proposed approach by using a set of specially designed memory-intensive applications and SPEC CPU2006 benchmarks.

Experimental Testbed.

Our emulation platform is implemented and evaluated on the dual-socket system with **Intel Xeon E5-2450** processor that supports up to 3 DDR3 channels and a total of 16 *two-way* hyper-threaded cores running at 2.1 GHz. Cache sizes of L1I, L1D, L2 and L3 are 32 KB, 32 KB, 256 KB and 20 MB respectively, and the total amount of DRAM is 32 GB.

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 3 shows the memory bandwidth measured using *copy* kernel of STREAM benchmark [5] 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.

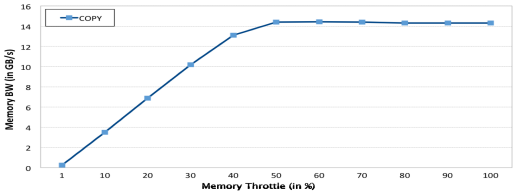


Figure 3: Relationship between memory throttling using thermal control registers and memory bandwidth of STREAM benchmark (*copy* kernel).

Approach for Validating Memory Latency Emulation.

Validating the memory latency model requires comparing the application performance as predicted by the performance model to the application performance as measured on real hardware with "physically slower" (higher-latency) memory. This validation is non-trivial as commodity hardware platforms do not support configuring different memory latencies. The lack of such hardware feature has served as a primary motivation for our software emulation approach.

To validate our model against physically increased memory latency, we leverage the different access latencies of local and remote DRAM in a multi-socket machine as follows. We create two different configurations for our experiments:

- *Conf_1* - a single socket is used for executing applications, i.e., processor and memory from the same socket;
- *Conf_2* - a processor from one socket and remote memory from the other socket are configured to run the same applications. We use the *numactl* tool 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.

First, we run a set of latency-sensitive experiments on *Conf_1* with *NVMpro* which injects software created delays based on the proposed memory latency model to mimic the latency of remote socket memory. Thus, we specify the *NVM latency* as the average latency to access remote socket memory. The *epoch size is 10 milliseconds* in all our experiments. We use Linux *perf* monitoring tool [2] to monitor the raw processor events needed by our latency model including memory stall cycles (LDM_STALL), and last level cache hit and miss ratios (LLC_HIT, LLC_MISS).

Then for validation and comparison, we measure application completion times executed on *Conf_2* (without *NVMpro*).

Applications and Benchmarks.

Pointer-Chasing Microbenchmark: We designed a memory-latency bound microbenchmark with a configurable degree of memory parallelism. The microbenchmark creates a pointer chain as an array of 64-bit integer elements. The contents of each element dictate which one is read next; 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 microbenchmark is *memory-latency sensitive* because the next element to be accessed is determined only after the current access completes. The microbenchmark can also create *multiple* independent chains to experiment with *different degrees of memory 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.

SPEC CPU2006 Bench: We use twenty applications from the SPEC CPU2006 [4] benchmark suite that offer a broad and representative coverage of real applications with diverse compute and memory characteristics.

Validating Accuracy of Memory Latency Emulation.

Pointer-Chasing Microbenchmark. Figure 4 compares the additional execution times of the pointer-chasing microbenchmark as estimated by our latency model (Eq. 2 used by the emulator) on *Conf_1* and the actual additional execution times measured by running the same benchmark on *Conf_2* compared against the base line execution on *Conf_1*)[2]. The X-axis reflects the memory level parallelism of the microbenchmark, which is defined by the number of independent memory accesses issued by microbenchmark at each iteration.

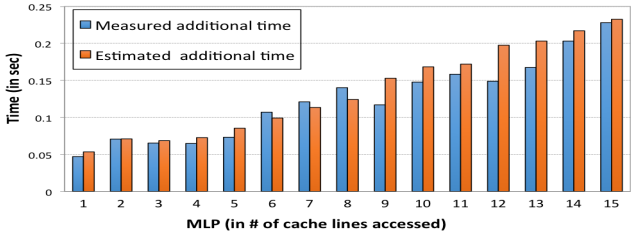


Figure 4: Comparison between the *measured* additional execution time in *Conf_2* vs *estimated* delay with the emulation platform in *Conf_1* for Xeon E5-2450-based system.

We observe that our model very closely matches the measured additional times. The average absolute error is 4.6%, and the minimum and maximum errors are being 0.16% and 11% respectively. This validates the correctness and high accuracy of the proposed model.

SPEC CPU2006 Benchmark. Figure 5 shows the LLC misses per 1000 instructions across the executed benchmarks in the suite. This signifies the *memory intensity* of different benchmark applications, i.e., how often they access the memory system. We can observe that *mcf*, *omnetpp*, and *milc* have a relatively high LLC miss rate, and they are memory-intensive, while applications like *hmmer*, *h264*, etc., have very low memory intensity and are rather compute-intensive.

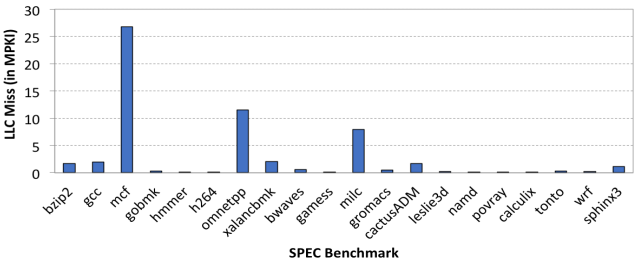


Figure 5: LLC miss of various SPEC benchmarks.

[2]All the experiments are performed five times, and the measurement results are averaged. This comment applies to all results in this section.

In order to formally evaluate the accuracy of the designed memory model we compute the *prediction error* as follows:

$$error = \frac{|CT^{remote}_{native} - CT^{remote}_{emulated}|}{CT^{remote}_{native}} \quad (3)$$

where CT^{remote}_{native} is the measured application completion time when benchmarks are executed (without the emulator) on the memory allocated in remote socket (i.e.,*Conf_2*), and $CT^{remote}_{emulated}$ is the measured application completion time when benchmarks are executed with *NVMpro* emulator on memory allocated in local socket (i.e., *Conf_1*) while **emulating** the memory latency of the remote socket.

Figure 6 depicts the accuracy results of the validation experiments. We can observe that for most benchmarks the errors for measured vs estimated execution times are low with an average and maximum errors of 1.8% and 5.36% respectively. As a sanity check, we also analyzed a variability of benchmarks' completion times without the emulator: the measured completion times are very consistent (less than 2% of corresponding average completion times).

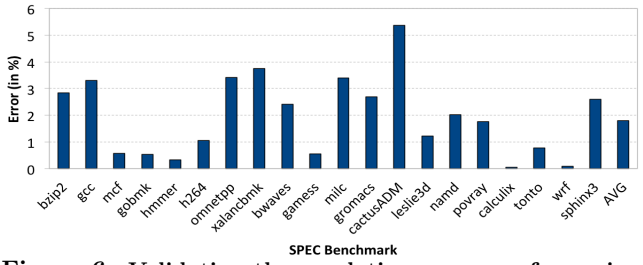


Figure 6: Validating the emulation accuracy for various SPEC benchmarks on Xeon E5-2450 system.

These experiments validate that the proposed memory model not only mimics the NVM performance specification but also achieves it with a high accuracy.

4. RELATED WORK

Several previous projects attempted to emulate performance of NVRAM using DRAM. Dulloor et al. [9] 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. [12] make use of *offline analysis* using the PIN binary instrumentation tool to estimate the average number of cache misses per program regions. Then they use the cache miss estimates to introduce additional delay during actual runs on bare metal. Instead, we focus on an *online model* that does not require the extra step of offline analysis.

Finally, Volos et al. [15, 14] emulate only NVRAM 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 NVRAM reads.

Previous work [11] has proposed that DRAM bandwidth throttling can be used 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, and there is a difficulty with achieving an accurate latency "slowdown" using this method. Moreover, this assumption is not true for latency-bound applications whose bandwidth requirements are very small. In contrast, our approach decouples latency emulation from bandwidth emulation.

There is a body of works that utilize performance counters for analyzing application memory performance. Green governor model [10] 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, this model ignores memory-level parallelism, which might lead to an over estimate of the actual memory time. Recent work [13], 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).

5. CONCLUSION

In this work, we introduce a novel platform, called *NVMpro* for emulating NVM systems with different performance characteristics. *NVMpro* offers two performance knobs for changing NVM's bandwidth and NVM's latency. The experiments with specially designed memory-intensive applications and SPEC CPU2006 benchmarks on the multi-socket machine show that the emulation platform supports a high degree of accuracy: the completion times of emulated applications are within 5% (on average) of the measured ones.

To control memory latency we implemented a software-based solution that injects a pre-computed delay in the stream of memory references to achieve a desirable (average perceived) NVM latency. The solution is based on the memory model that leverages hardware performance counters. The counters are read with *perf* tool at specified time intervals to dynamically compute the additional delay to inject. As the *perf* tool uses a system call to monitor the events, there is an overhead associated with crossing the user-kernel protection boundary. To minimize this overhead, we are exploring an alternative implementation for reading performance counters directly via RDPMC instruction [3]. Our next steps also include extending the emulator for multi-threaded applications by taking into account synchronization primitives.

6. REFERENCES

[1] 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`.

[2] Perf. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[3] RDPMC– Read Performance Monitoring Counters. `http://www.rcollins.org/p6/opcodes/RDPMC.html` .

[4] SPEC CPU2006. `https://www.spec.org/benchmarks.html`.

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

[6] 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`.

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

[8] 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.

[9] S. R. Dulloor, S. Kumar, A. Keshavamurthy, , P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proc. of EuroSys*, 2014.

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

[11] H. Hanson and K. Rajamani. What Computer Architects Need to Know about Memory Throttling. In *Proc. of ISCA*, 2010.

[12] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVM Era. In *Proc. of PVLDB*, 2013.

[13] 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.

[14] 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.

[15] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS 16*, ASPLOS '11, 2011.