Proteus: A Flexible and Fast Software Supported Hardware Logging approach for NVM

Seunghee Shin Satish Kumar Tirukkovalluri James Tuck Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{sshin6,stirukk,jtuck,solihin}@ncsu.edu

ABSTRACT

Emerging non-volatile memory (NVM) technologies, such as phase-change memory, spin-transfer torque magnetic memory, memristor, and 3D Xpoint, are encouraging the development of new architectures that support the challenges of persistent programming. An important remaining challenge is dealing with the high logging overheads introduced by durable transactions.

Here, we propose a new logging approach, *Proteus*, for durable transactions that achieves the favorable characteristics of both prior software and hardware approaches. Like software, it has no hardware constraint limiting the number of transactions or logs available to it, and like hardware, it has very low overhead. Our approach introduces two new instructions and hardware support, primarily within the core, to manage the execution of new instructions. We also propose a novel optimization at the memory controller that is enabled by a persistent write pending queue in the memory controller. We drop log updates that have not yet written back to NVMM by the time a transaction is considered durable.

We compared our design against state-of-the-art hardware logging, ATOM [3], and a software only approach. Our experiments show that *Proteus* improves performance by 1.44-1.47× depending on configuration compared to a system without hardware logging and 9-11% faster than ATOM. A significant advantage of our approach is dropping writes to the log when they are not needed. On average, ATOM makes 3.4× more writes to memory than our design.

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies, such as phase-change memory, spin-transfer torque magnetic memory, memristor, and 3D Xpoint, are now available in the market. For example, 3D Xpoint memory is already in the market since 2017 [2]. Due to their non-volatility and byte addressability, a subset of them which have low read latencies are being considered for use as main memory, either to augment or replace DRAM. A *non-volatile main memory* (NVMM) can be directly accessed using load and store instructions. This gives an opportunity for programmers to persist important data in data structures in main memory, skipping the need or overheads of serializing it to the file system.

Persisting data structures in main memory must be implemented in a way that ensures data consistency, so that software can recover to a consistent state in the event of software or hardware failures. Data consistency is difficult to ensure in systems with volatile caches because the order in which values are *persisted* (e.g. written back to the NVMM) depends on the cache replacement policy, which often differs from program order. To deal with these ordering challenges, architects have proposed memory persistency models [4] to provide programmers a guarantee of the ordering in which stores are persisted to NVMM.

One model that is easier for programmers to reason about is that of *durable transactions* [5]. With a durable transaction, all stores in a transaction persist together or none of them do. This is a simple and useful abstraction for programmers.

We explore a key challenge of using durable transactions: how to perform logging efficiently and flexibly. Durable transactions require logging, either through *redo* or *undo* logging. The log allows the transaction to be recovered if a failure occurs during the transaction. Each log entry can be created through software code inserted by the programmer, through a library [1], or directly in hardware without additional code [3]. Software approaches (SW) incur large performance overheads due to additional instructions but offer the greatest flexibility, including unlimited transaction size and control over logging operations. The latter approach (HW) [3] has low performance overheads, but it is typically less flexible.

2 SOFTWARE SUPPORTED HARDWARE LOGGING

We propose a new logging approach, referred to as *Proteus*, that achieves the favorable characteristics of both software and hardware approaches: software controls it's own log area, it can support unlimited transactions of arbitrary size, it can manage its own recovery, and it has low overhead. Our approach introduces two new instructions that a log-load instruction creates a log entry by loading the original data, and a log-flush instruction writes the log entry to NVMM. We presume no additional programming effort beyond specifying transaction boundaries, since the compiler can generate instructions appropriately for code inside transactions.

Additional hardware support is introduced, largely within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data to ensure durable transaction semantics. *Proteus* avoids any limitation on the size or number of transactions through judicious design of the interface: software remains in control of allocating the log area, and hardware keeps the cost of updating the log low.

LogQ is a structure that keeps track of each logging operation. When a log-flush instruction enters the OOO pipeline, an entry is created in the LogQ. It contains the log-from address (location of the original data in NVMM), log-to address (location of the log entry in the log area), and log-data (data value to be flushed to NVMM). When the log-flush is received at the memory controller

Note a fundamental difference between durable transaction and transactional memory (TM): a durable transaction specifies when data is made durable in NVMM, whereas TM deals with when data is visible to other threads. Consequently, durable transactions apply even to sequential code.

(MC), the MC sends an acknowledgment to the LogQ and the entry is deallocated from the LogQ. The LogQ has another important function: imposing ordering between a log-flush instruction and a store to the same log-from address. Thus, when a store retires and before it's committed to the cache, it checks its address against older entries in the LogQ. To achieve higher performance, the LogQ allows log entries to flush out-of-order. Furthermore, the LogQ can hide the latency of logging by enabling the concurrent execution of the actual flushes to the MC.

Log Lookup Table (LLT) keeps the last few log-from addresses in a transaction and prevents repeated logging operations to the same log data, reducing the memory bandwidth devoted to logging. In addition, the LLT helps to reduce the size of the log area in NVMM, the LogQ, and the LPQ in the MC. When a transaction ends, triggered by the tx-end instruction, the LLT is cleared.

We also consider integrating *Proteus* with a battery backed WPQ, allowing the WPQ to be considered part of the persistency domain. Once writes reach the WPQ they are considered durable. The presence of a battery-backed WPQ is consequential: it presents a new opportunity to avoid writes to the NVMM. A key observation which we exploit is that most log entries are created and discarded, because failures are rare. Thus, we add LPQ which operates the same as WPQ, but log-flushes go only to the LPQ. This distinction allows us to treat them differently, where log entries are kept as long as possible in the LPQ and discarded when a transaction commits.

3 EVALUATION

For our experiments, we implemented Intel PMEM instructions, *clwb*, *clflushopt*, and *pcommit* in a processor simulator built on MarssX86, which is an open source cycle-accurate full system simulator for an x86-64 architecture.

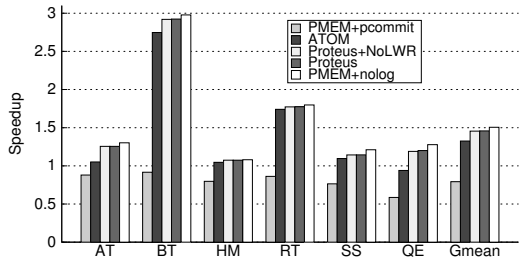

Figure 1: Speedup comparison on NVMM, with software logging with PMEM as baseline.

In order to assess the performance benefits of *Proteus*, we implemented and compared the following schemes: software logging represented by an Intel PMEM based implementation of WAL, both with *pcommit* (PMEM+pcommit) and without it (as the base case), hardware logging represented by ATOM [3] including all of its optimizations (ATOM), and software-supported hardware logging represented by our scheme *Proteus* (Proteus) and without log write removal (Proteus+NoLWR). In order to see how close they perform to an ideal case, we also implemented PMEM but with logging removed (PMEM+nolog). The latter does not provide failure safety and is devoid of any logging overheads, and thus it is an ideal case.

The result of their speedup over the base case of PMEM without pcommit for all benchmarks and for the geometric mean of all benchmarks are shown in Figure 1. First, let us observe the PMEM+pcommit bars. They are significantly below 1.0 in all benchmarks, with a geometric mean of 0.79. This shows that moving the MC and WPQ into the persistency domain is very helpful for performance. Next, consider the last bars (PMEM+nolog) that are significantly higher than 1.0, with a geometric mean of 1.51. This shows that the addition of logging code and its execution causes very significant performance overheads, whereas its removal speeds up execution by 51% on average. Now let us examine ATOM and *Proteus*. ATOM performs quite well, achieving a 1.33× speedup on average while *Proteus* achieves a geometric average of 1.46× speedup. In other words, *Proteus* is faster than ATOM by $\frac{1.46}{1.33} - 1 = 10\%$. Furthermore, *Proteus*'s speedup is only 3.3% lower than the ideal case of no logging.

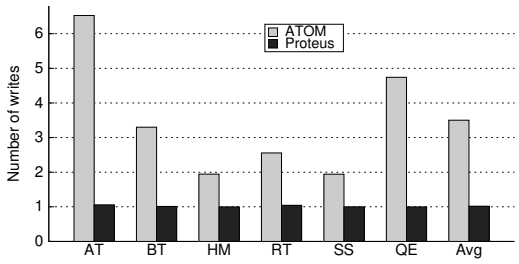

Figure 2: The number of NVMM writes, normalized to PMEM with no logging.

Figure 2 compares the number of NVMM writes for each benchmark, normalized to the number of NVMM writes of PMEM+nolog. On average, ATOM has three times more writes to NVMM (3.4×), compared to PMEM without logging. In benchmark (QE), it more than quadruples the writes to NVMM and in the worst case (AT), it has six times more writes to NVMM. The increase in number of writes is due to logging (creation and truncation). This is significant because it cuts the write endurance of NVMM by more than three quarters. In contrast, *Proteus* only increases the number of writes slightly. In the worst case (AT), the increase in writes is still relatively low, at 6%. The reason for *Proteus*'s advantage is that most log updates are held at the LPQ and flash cleared when a transaction ends, thanks to the fact that the MC is part of the persistency domain. Thus, most log flushes do not even go to the NVMM.

REFERENCES

[1] NVM Library Team at Intel. 2016. Persistent Memory Programming. (August 2016). http://pmem.io.

[2] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology. (Jul. 2015). https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology

[3] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. DOI: http://dx.doi.org/10.1109/HPCA.2017.50

[4] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[5] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 178–190. DOI: http://dx.doi.org/10.1145/3123939.3124539