Persistent Memory Analysis Tool (PMAT)

Louis Jenkins and Michael L. Scott
Computer Science Department, University of Rochester
{louis.jenkins@rochester.edu, scott@cs.rochester.edu}

***Abstract*— Intel's Persistent Memory Development Kit (PMDK) provides two separate tools—pmreorder and pmemcheck—that allow the programmer to explore and verify the consistency of possible states that may be present in persistent memory in the wake of a crash, given that caches may write back values out of order. Unfortunately, these tools are heavy-weight and inefficient, and so we have developed the Persistent Memory Analysis Tool (PMAT), derived from pmemcheck, that enables the simulation and verification of hundreds to thousands of crashes per second for programs of significant size, for an improvement of at least three orders of magnitude.**

I. Introduction and Related Work

Intel's Persistent Memory Development Kit (PMDK) [5] provides two tools to verify the crash consistency of persistent applications. The first of these, *pmemcheck*, is a plugin for Valgrind [2]. It traces all stores and flushes to marked persistent regions, together with all store/memory fences. Valgrind performs dynamic recompilation on-the-fly, inserting desired instrumentation. In comparison to source-based tools, recompilation significantly increases coverage, encompassing even third-party binary-only libraries. Valgrind tends to be significantly slower than normal execution, not only due to the cost of recompilation and of instrumentation, but also because it serializes multi-threaded applications, running only one thread at a time. At the same time, it introduces context switches with sufficient frequency and irregularity to manifest most of the races that would occur during normal execution.

The pmemcheck tool uses Valgrind to trace at the granularity of individual stores. It keeps these stores (and flushes and fences) in a somewhat expensive internal structure that allows it to check for potentially erroneous idioms such as repeated (unflushed) stores to the same persistent location. More problematically, for programmers interested in consistency, it logs the trace to a file for subsequent perusal by *pmreorder*, the second tool mentioned above. Logging leads to both an I/O bottleneck and, for programs of realistic size, potentially enormous files.

The *pmreorder* tool processes a pmemcheck trace and generates alternative store orders that are consistent with the flushes and fences. These are orders that could have occurred due to out-of-order writeback from the last-level cache. Because the set of possible reorders is exponentially large, pmreorder allows the user to choose from among a variety of "reordering engines," which embody various sampling strategies. Each sampled ordering is then used to create a memory image that is passed to a user-provided verification (consistency-checking) function. The overall goal—sampling possible memory images and checking them for consistency—is hampered both by the separation of trace generation and reordering and by inefficiencies in the reordering mechanism. To address both of these problems, we introduce a *Persistent Memory Analysis Tool* (PMAT) that integrates consistency checking into pmemcheck, eliminating the need for tracing. The result is a dramatic increase in performance.

This work was supported in part by NSF grants CCF-1422649, CCF-1717712, and CNS-1900803; by a Google Faculty Research award; and by a US Department of Energy Computational Science Graduate Fellowship (grant DE-SC0020347).

II. Design and Implementation

Unlike the original pmemcheck, PMAT aggregates stores to persistent locations into fixed sized cache lines, which are then fed to a bounded, software-simulated *CPU cache* and *reorder buffer*. When the number of written lines exceeds the cache capacity, lines are evicted at random. They are also evicted in response to flushes in the application (and written back without evicting in response to write-back instructions). Evicted and written-back lines are fed to the reorder buffer, which can in turn randomly permute its in-flight lines subject to limitations imposed by fence instructions. Lines that graduate out of the reorder buffer are written to a *shadow heap* of the same size as the persistent memory region originally declared by the user. In order to capture all stores to persistent locations, regardless of where they occur in the source code, we currently instrument all stores and filter them by address. While this is expensive, we see no practical alternative in the absence of language and compiler support for persistence.

Whenever a store or flush is made to an address inside a persistent memory region, and before and after any fence is performed, PMAT chooses, with small probability, to simulate a crash and to pass the contents of the shadow heap to a user-provided verification function, much as in pmreorder. A user may also explicitly simulate a crash, and may even disable the randomized crash simulation—e.g., when initializing data, or to incrementally introduce the testing tool to the code base to avoid simulating crashes during parts of execution not yet supported by the verification function.

The verification function, provided by the user as a standalone program, takes as argument the name of a file in which to find the shadow heap. As an optimization, the user may mark portions of the persistent memory as "transient," in response to which PMAT will ignore stores to that region, on the assumption that it will also be ignored by the verification function. For use in source-code assertions, PMAT also allows the user to query whether a line is currently dirty in the simulated cache or re-order buffer (implying it has not been written back and fenced).

If a verification fails, the user is provided the contents of stderr, stdout, and the stack trace for the last store to each cache line that has not been flushed or fenced, each as their own file. This information can significantly aid in determining the root cause of the problem in the application. Also provided is the shadow heap itself, which can be further analyzed and even loaded into memory.

```
#include <valgrind/pmat.h>

// Persistent Memory Region API
PMAT_REGISTER("dummy.bin", addr, len);
PMAT_UNREGISTER_BY_ADDR(addr);
PMAT_UNREGISTER_BY_NAME("dummy.bin");
PMAT_TRANSIENT(addr, len);
PMAT_IS_DIRTY(addr, len);

// Crash Simulation API
PMAT_CRASH_DISABLE();
PMAT_CRASH_ENABLE();
PMAT_FORCE_CRASH();
```

III. Experiments

The current release of pmreorder includes a single example application, called pmreorder_list [1]. This example embodies a singly linked list that has been deliberately designed to perform an arbitrary number of inconsistent (i.e., incorrect from the perspective of persistence) insertions. We use this example to compare (Figure 1) the performance of PMAT to that of pmemcheck + pmreorder, with the latter using Intel's comparatively lightweight ReorderPartial engine. To highlight the constituent costs of the Intel tools, we also show individual times for pmemcheck and pmreorder. For a given list size, the costs of pmemcheck and pmreorder sum to that of the combination, but pmreorder overwhelmingly dominates (orange and red dots overlap in the figure).

Given limited time for experimentation, we set a timeout of 5 hours on every program run. The pmreorder tool reaches this limit at a list size of only 128K nodes. The much faster pmemcheck is still able to accommodate only 512K nodes. PMAT, by contrast, can process a list of 1M nodes in just under a half an hour when set to simulate a crash with probability 0.01 after each store, flush, and before and after each fence. During the half-hour run, it simulated over 62 thousand crashes—41 of them per second. Approximately 1/3 of the crashes revealed an inconsistency. Calls to the verification function consumed a mean time of 5.88ms each.

By design, pmreorder_list flushes stores out of order: the value of each new node is flushed *after* it is inserted into the list. PMAT has no trouble discovering this error. Interestingly, the frequent flushes mean that the benchmark doesn't actually stress the simulated cache and write buffer.

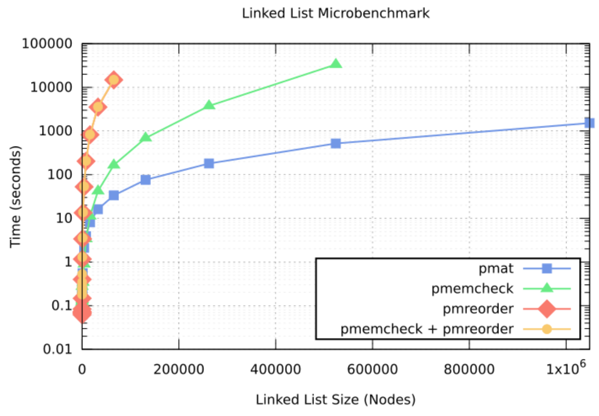

Fig. 1. pmat vs pmemcheck + pmreorder

In our experience, pmemcheck and pmreorder have been too slow to use in any larger, more realistic application. To further test PMAT, however, we have experimented with the durably linearizable queue of Friedman et al. [3]. Manually injected errors in this application were found in a matter of minutes.

IV. Future Work

While the performance of PMAT is far superior to that of pmemcheck + pmreorder, there are multiple areas for improvement and enhancement. The thread serialization inherent in the Valgrind virtual machine means that only one core will be active in a given test. This suggests the possibility, on a multicore machine, of running verification (or multiple verifications) in parallel with the application. Furthermore, while PMAT already allows the programmer to query whether a line has been persisted, it may be worthwhile to provide additional assertions on ordering constraints in a manner similar to PMTest [4]. Given the inherent difficulty in writing verification checkers for larger applications, this approach may allow certain classes of errors to be found with less programmer effort.

References

[1] pmreorder GitHub repository. https://github.com/pmem/pmdk/tree/stable-1.7/src/examples/pmreorder.

[2] Valgrind. https://valgrind.org/.

[3] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proc. of the 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, Vienna, Austria, 2018.

[4] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proc. of the 24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–425, Providence, RI, March 2019.

[5] Usharani U. and Andy M. Rudoff. Introduction to programming with Intel Optane DC persistent memory. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel, August 2017.