NVthreads: Practical Persistence for Multi-threaded Applications *

Terry Ching-Hsiang Hsu[1], Helge Brügner[2], Indrajit Roy[3], Kimberly Keeton[4], Patrick Eugster[1,5]
Purdue University[1], TU Münchun[2], Google[3], Hewlett Packard Labs[4], Università della Svizzera italiana[5]

1. Introduction

Recently proposed frameworks for persistent programming provide multiple ways to directly manipulate data structures stored in non-volatile memory (NVM) [6, 7, 14]. Application developers can either rewrite their program to use durable transactions (NV-Heaps [7], Mnemosyne [14]) or rely on the compiler and runtime system to infer failure atomic regions from locks (Atlas [6]). These approaches track persistent data at a very fine granularity, such as at the level of individual store operations, and use cache flushes and write-ahead logging to correctly recover from failures. Unfortunately, the high overheads of tracking, logging, and managing volatile caches in these systems result in a huge overhead – sometimes an order of magnitude slowdown between unmodified DRAM based applications and their crash tolerant versions. Certain systems propose hardware modifications to ameliorate the cost of cache flushes and ordering of NVM writes but do not work on today's processors [7, 8].

Our goal is to provide a simple transition path for existing C/C++ programs to leverage non-volatile memory. We want to enable the use of NVM with few or no program modifications, and yet have good performance on today's processors. Our key observation is that, for many applications, the high overheads of maintaining logs can be reduced substantially by using coarse-grained tracking, such as at the level of memory pages.

2. Programming Model and Design

We propose NVthreads, a programming model and system that adds durability guarantees to existing `pthreads` based multi-threaded C/C++ programs. NVthreads executes a multi-threaded program as a multi-process program using DThreads [11] to buffer intermediate changes with virtual memory when data structures may be in an interim state. When program data is in a consistent state, NVthreads commits modified memory pages to a durable log for recovery. By using the operating system's copy-on-write mechanism, NVthreads can efficiently buffer uncommitted writes and requires only a redo log to recover. NVthreads builds on the observation that synchronization operations, such as lock acquire and release, provide enough information to determine the boundaries of failure atomic regions [6] for data consistency guarantees. Thus, instead of requiring programs to be re-written with durable transactions, NVthreads adds durability semantics by automatically inferring when data is safe to write to persistent memory.

Example. Figure 1 shows an implementation of K-means that becomes crash tolerant when linked with NVthreads. K-means is a clustering technique that divides the input dataset (stored in the array `points`) into `K` groups. The algorithm proceeds in rounds, refining the centers until convergence. In an iteration, each point is first assigned to the closest center (stored in the array `labels`), and then centers are updated by taking the average of points assigned to them. In lines 3-10 the centers are first initialized using the current labels. In each

```
1  //points: input points
2  //labels[i]: id of center closest to point i*/
3  float* kmeans(float* points, float* labels){
4    centers = calculateCenters(labels);
5    while (!converged){
6      pthread_create(.., findDistance, ..);
7      pthread_join(..);
8      centers = updateCenters(labels);
9    }
10   return centers;}
11 void findDistance(points, labels, centers){
12   pthread_lock(&m_xy); //find closest center point [X,Y]
13   labels[X:Y] = closestCenters[..];
14   pthread_unlock(&m_xy);
15   ...}
16 void main(){
17   if (crashed()) //Recovery code
18     nvrecover(labels, N*2*sizeof(float),'labels');
19   else //Only 'labels' is persistent
20     labels=(float*)nvmalloc(N*sizeof(float),'labels');
21   ...
22   ans = kmeans(points, labels);}
```

Figure 1: Pseudo-code for multi-threaded K-means.

iteration threads calculate the closest center to a subset of points, and update the corresponding labels in a critical section (lines 12-14). When a thread exits the critical section NVthreads guarantees that the labels of all the points it was working on have been updated. Lines 17-18 depict the recovery code. After a crash, the program re-reads `labels` from NVM. Otherwise, it allocates memory in NVM to store the labels. The real recovery work is performed by `nvrecover` which is application agnostic and implemented in the NVthreads runtime. After a crash, the K-means algorithm will simply restart its execution by calculating the latest centers from `labels`. Therefore, the time to converge will decrease compared to starting from scratch.

Techniques. In a bit more detail, NVthreads works as follows: it (1) uses critical sections to determine failure atomic regions, (2) tracks dependence between failure atomic regions to decide when to make logs permanent, (3) uses redo logs to ensure NVM data is consistent after a crash, and (4) runs the optional application specific recovery code before resuming execution. Table 1 summarizes the benefits of NVthreads' design decisions over existing persistent programming systems. In stark contrast to systems that track durable data at the level of individual words [14] or stores [6], NVthreads reduces the overheads of ordering writes to NVM by removing the need to flush data after each write, and requiring that only log entries be ordered.

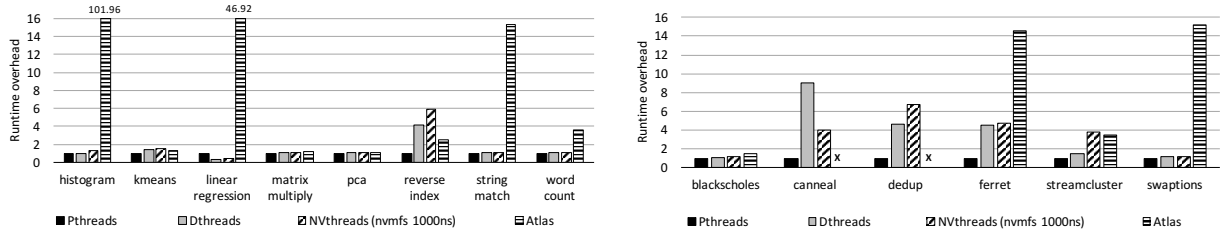
3. Evaluation

We evaluated NVthreads on a wide range of benchmarks and real-world applications on a Ubuntu 14.04 (Linux 3.16.7) server with two Intel Xeon X5650 processors (12 cores@2.67 GHz), 198GB RAM, and 600GB SSD. We used an NVM emulator based on a modified `tmpfs` on DRAM that injects 1μs latency for each 4KB write. For an emerging class of workloads depicted in the PARSEC [4] and Phoenix [13] suites, NVthreads performs as well as, and in most cases significantly better, than

*https://dl.acm.org/citation.cfm?id=3064204 [9]

Issue	Commonly used solutions	NVthreads approach	NVthreads advantage
Determine consistency points	Use durable transactions [7, 14]	Infer from synchronization points	Ease of programming
Handle transaction aborts	Use undo logs [6, 12], word level STM [14]	Use process memory, no undo logs	Lower overheads, simpler recovery
Handle in-flight updates	Use redo logs	Use redo logs	None
Track updates	Intercept each store [6] or word [14]	Intercept page writes	Amortized costs, good performance
Volatile caches	Flush writes, some require new hardware [7, 8, 10]	Flush log entries (memory pages)	Amortized costs, no processor changes

Table 1: NVthreads design decisions that improve performance and programmability.



(a) Phoenix applications (b) PARSEC applications

Figure 2: Cost of crash tolerance. Lower is better.

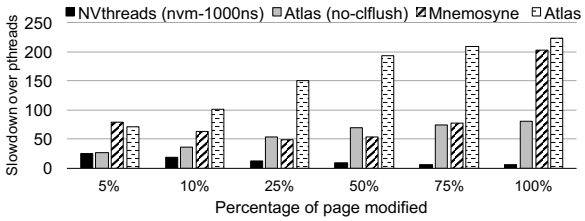


Figure 3: Benefits of NVthreads. Lower is better.

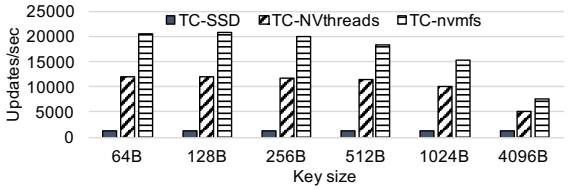


Figure 4: Throughput of Tokyo Cabinet. Higher is better.

Atlas (Figure 2). The DThreads performance numbers give a sense of the overheads of converting multi-threaded execution into multi-process execution. We compare NVthreads with Mnemosyne [14] and Atlas [6] that use word or store level data tracking using a microbenchmark that allocates 1000 memory pages (4MB of data), and then 4 threads modify a random region of each page. Both Mnemosyne and Atlas are 70× slower than `pthreads` in most cases and more than 200× slower when threads modify 100% of each page (Figure 3). Since tracking data at page granularity allows NVthreads to amortize logging overhead as a bigger fraction of each page is modified, NVthreads is 3-30× faster than these systems. We also compare these systems on PageRank, which iteratively determines the importance of nodes in a Web graph [5]. We use the real-world Slashdot graph dataset which has 82K vertices and 950K edges [1]. NVthreads is more than 2× faster than Atlas and 10× faster than Mnemosyne at 12 cores. We run NVthreads and Atlas on much larger graphs such as the 1.2 GB Livejournal data [3]. NVthreads completes each PageRank iteration in 5s with twelve threads and is almost 6× faster than Atlas which takes 29s. The `pthreads` version takes less than a second per iteration. For crash recovery, our evaluation on K-means shows that NVthreads helps applications converge up to 2× faster after a failure versus their counterparts that need to start from the beginning. For key value stores, we configured Tokyo Cabinet, a high performance library for database management [2], to make its B+-tree durable by linking it with NVthreads. This experiment shows that NVthreads can be integrated without making any changes to the B+-tree. When we use NVthreads to make the B+ tree in Tokyo Cabinet durable (*TC-NVthreads*), its throughput on the NVM emulator is 4×-9× better than using unmodified Tokyo Cabinet on SSDs (*TC-SSD*). Tokyo Cabinet running on our NVM emulator (*TC-nvmfs*) is about 30%-50% better than TC-NVthreads (Figure 4). NVthreads incurs higher overheads but avoids the code complexity of a custom transaction system.

4. Conclusion

NVthreads uses fast NVM to make C/C++ programs fault tolerant. It leverages synchronization operations to determine consistency semantics, and coarse-grained page-level tracking to manage durable data. Compared to state-of-the-art persistent systems, NVthreads significantly reduces the performance gap between unmodified applications and their crash tolerant versions. Overall, given the familiar interface of a multi-threading library, we believe NVthreads is a design point which makes it simpler for programmers to transition to the NVM era.

References

[1] Stanford network analysis package. http://snap.stanford.edu/snap.

[2] Tokyo Cabinet. http://fallabs.com/tokyocabinet/.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. KDD '06, 2006.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. PACT 2007.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW7*, 1998.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. OOPSLA '14, 2014.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. ASPLOS XVI, 2011.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-addressable, Persistent Memory. SOSP '09, 2009.

[9] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. EuroSys '17, 2017.

[10] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. ASPLOS '16, 2016.

[11] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. SOSP '11, 2011.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1), Mar. 1992.

[13] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. HPCA '07. IEEE Computer Society, 2007.

[14] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. ASPLOS XVI. ACM, 2011.