NVthreads: Practical Persistence for Multi-threaded Applications

Terry Ching-Hsiang Hsu
Purdue University
terryhsu@purdue.edu

Helge Brügner
TU München
helge.bruegner@tum.de

Indrajit Roy
Google Inc.
indrajitroy@google.com

Kimberly Keeton
Hewlett Packard Labs
kimberly.keeton@hpe.com

Patrick Eugster
TU Darmstadt and Purdue University
p@dsp.tu-darmstadt.de

Abstract

Non-volatile memory technologies, such as memristor and phase-change memory, will allow programs to persist data with regular memory instructions. Liberated from the overhead to serialize and deserialize data to storage devices, programs can aim for high performance and still be crash fault-tolerant. Unfortunately, to leverage non-volatile memory, existing systems require hardware changes or extensive program modifications.

We present NVthreads, a programming model and runtime that adds persistence to existing multi-threaded C/C++ programs. NVthreads is a drop-in replacement for the `pthreads` library and requires only tens of lines of program changes to leverage non-volatile memory. NVthreads infers consistent states via synchronization points, uses the process memory to buffer uncommitted changes, and logs writes to ensure a program's data is recoverable even after a crash. NVthreads' page level mechanisms result in good performance: applications that use NVthreads can be more than 2× faster than state-of-the-art systems that favor fine-grained tracking of writes. After a failure, iterative applications that use NVthreads gain speedups by resuming execution.

1. Introduction

Memristor [33], phase-change memory [23], and other emerging non-volatile memory (NVM) technologies will provide disk-like persistence but at latency as low as main-memory (DRAM) devices [3]. These NVM devices will be accessible by memory instructions, and will result in high performance, fault tolerant programs that avoid the overheads of traditional persistent media, such as deep software layers and the cost of serializing and storing data. Programs may even eliminate the distinction between in-memory versus on-disk representations of data. Recent partnership announcements from Intel-Micron [3] and HPE-SanDisk [4] aim to bring this memory-centric computing to consumers. Unfortunately, to fully benefit from low latency persistence, developers need to re-architect both system and application software [9].

When manipulating persistent data directly, applications need to ensure that failures during updates don't end up corrupting data. As an example, a failure during insertion of an element to a persistent linked list should not result in dangling pointers or other corruptions. A safe way to manipulate persistent data is to ensure that data structure updates are failure atomic, i.e., even in the presence of failures, either all or none of the updates are reflected in the NVM. The challenge in implementing failure atomicity is to correctly handle partial updates even in the presence of multi-threading, volatile caches, and reordering of NVM writes by the processor. Managing persistent data structures is costly due to these challenges (e.g., frequently flushing cache lines to NVM is very costly [37]). For any persistent programming system to be practical, its overheads should be low enough that applications actually benefit from using NVM.

Recently proposed frameworks provide multiple ways to directly manipulate NVM data structures [14, 16, 34]. Application developers can either rewrite their program to use durable transactions (NV-Heaps [16], Mnemosyne [34]) or rely on the compiler and runtime to infer failure atomic regions from locks (Atlas [14]). These systems track persistent data at a very fine-granularity, such as at the level of individual stores, 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 results in a huge performance gap, sometimes an order of magnitude slowdown, between un-

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064204

modified DRAM based applications and their crash tolerant versions (Section 5.6). Certain systems propose processor modifications to ameliorate the cost of cache flushes and ordering of NVM writes but these systems do not work on today's processors [16, 17].

Our goal is to provide a simple transition path for existing C/C++ programs to leverage non-volatile memory. We want applications to use 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 redo logs in combination with coarse-grained tracking, such as at the level of memory pages.

We propose NVthreads, a programming model and system that adds durability guarantees to existing multi-threaded C/C++ programs. NVthreads uses two techniques to provide failure atomicity. First, NVthreads executes a multi-threaded program as a multi-process program, using virtual memory to buffer intermediate changes when data structures may be inconsistent. 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 (unlike the costly software transactional memory approach in Mnemosyne [34] or eager cache flushes in Atlas' undo logs [14]), and requires only a redo log to recover. Second, it builds on the observation that synchronization operations, such as lock acquire and release, provide enough information to determine the boundaries of failure atomic regions [14]. 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. While NVthreads' design can also be used to implement durable transactions, our current approach of using locks to infer consistency boundaries means that programs require very few modifications to start using NVM.

NVthreads uses multiple techniques to ensure good performance. Its approach of using virtual memory to track data structure modifications is in stark contrast to recent systems that track durable data at the level of individual words [34] or stores [14]. NVthreads reduces the overheads of ordering writes to NVM by eliminating the need to flush data after each program write, and requiring that only log entries be ordered. Even though NVM will be byte-addressable, our evaluation shows the importance of *coarse-grained tracking* of program writes, i.e., at the level of 4KB memory pages, for good performance. In fact, NVthreads is 2×–10× faster than Mnemosyne [34] and more than 2× faster than Atlas [14], both of which use fine-grained memory management. Additionally, NVthreads uses less space to store metadata and logs compared to these systems.

For many workloads in the PARSEC [11] and Phoenix [29] benchmarks NVthreads incurs modest overheads while making data structures durable. We use an emulator to show that NVthreads' performance results are robust even if NVM devices are much slower than DRAM. By using NVthreads and NVM, the persistent versions of programs are 15% to 22× faster than using solid disk drives to store logs. Our evaluation on a K-means clustering program shows that NVthreads helps programs converge up to 1.9× faster after a failure versus their non-durable counterparts. Finally, we integrate NVthreads with Tokyo Cabinet, a high performance key-value store, and show that for a modest increase in overheads it provides the same durability guarantees and without the need for custom transactional code.

The contributions of this paper are:

- A programming model and runtime that infers when data structures are consistent and adds durability semantics with very few program modifications.
- Novel mechanisms that use process memory to buffer uncommitted writes and track data at memory page level, thus avoiding undo logs and the need to instrument each program write.
- Extensive evaluation that shows that the NVthreads' approach has low overheads, and outperforms Mnemosyne and Atlas. Iterative applications require only tens of lines of recovery code and converge faster by resuming after failures.

The NVthreads source code is publicly available at:
https://github.com/HewlettPackard/nvthreads

2. Challenges in using non-volatile memory

We first review basic characteristics of NVM.

2.1 Non-volatile memory

Non-volatile memory (NVM) devices retain data even after a power loss, yet have access characteristics similar to DRAM and higher density than DRAM. New NVM technologies such as PCM [23], memristor [33], and STT-MRAM [18], will have access latencies similar to DRAM, which is three orders of magnitude faster than flash. Unlike flash and disks, these NVM devices will also be byte addressable, meaning they can be accessed through memory instructions, rather than requiring block-granularity read and write operations. Given these advantages, it is conceivable that NVM devices will not only be deployed as PCIe-attached devices (replacement of SSDs) but also be directly attached to the memory bus (similar to DRAM). However, even in such architectures we expect CPU caches to be volatile, and DRAM and NVM to coexist. While current applications will continue to work on NVM devices, they will not automatically take advantage of the low latency durability.

2.2 Issues

There are multiple challenges that need to be solved before programs can use NVM for fault tolerance.

```
1  // L is a persistent list
2  readFromNVM(&L);
3  ...
4  // Add element to the tail of list
5  pthread_lock(&m);
6  e = nvmalloc(sizeof(elem), 'e');
7  e->val = localVal;
8  tail->next = e;
9  e->prev = tail;
10 tail = e;
11 pthread_unlock(&m)
```

Figure 1: Pseudo-code that appends to a persistent list.

Data structure consistency. NVM-aware solutions need to ensure data consistency even in the presence of failures. Consider Figure 1 where a multi-threaded program adds an element to a doubly linked list. If a crash occurs during insertion, it is possible that the new element's back pointer is not set correctly, thus leaving the list inconsistent. As memory is persistent, after the crash, the program will not be able to proceed correctly.

Volatile caches. Even though programs will be able to directly access NVM, the underlying hardware may cache data in the volatile CPU caches, and reorder stores to NVM. Unfortunately, there is no easy and efficient way to determine what data has been persisted. In Figure 1, even if a crash occurred after the program executed line 10, the updates may not have reached the NVM, and the list could be inconsistent. Therefore, NVM-aware systems need to manage the movement of data from volatile caches to NVM.

Performance and programmability. Since objects in the heap will reside in the NVM, programs no longer need to bear the overheads of serializing and storing persistent data in filesystems. Still, there are performance and programmability costs associated with keeping heap objects consistent. For example, a possible solution to correctly handle state in volatile caches is to flush cache lines after each write, but such heavy-handed approaches result in high overheads. Some prior solutions mitigate these high overheads by assuming the presence of modified processors [16, 17], while others require extensive program modifications.

2.3 Related Work

Persistent programming models. BPFS [17] and PMFS [19] are example filesystems that leverage the low latency of NVM to accelerate filesystem operations. BPFS uses optimized shadow copying to maintain consistency, but requires new hardware primitives in the form of epoch barriers. These systems do not require any application changes, but restrict applications to the block based file system interface.

Mnemosyne [34] and NV-Heaps [16] expose direct NVM access but require applications to be rewritten with transactions. NV-Heaps relies on processor changes and maintains undo logs. NVthreads has similar goals as these systems, but uses multi-process execution, and does not require applications to use transactions. Mnemosyne extends software transactional memory with durability semantics and, similar to NVthreads, uses redo logs. However, our evaluation shows that NVthreads outperforms Mnemosyne by 2-10× because of its design choices and use of operating system techniques.

Prior work on Java concurrency control has shown how transactional boundaries can be inferred from locks [35]. Atlas extends this idea to add durability semantics to lock-based programs [14]. Atlas provides a compiler and runtime that instruments writes to NVM and creates an undo log for each store to aid recovery. JUSTDO logging improves performance and log management in Atlas like systems by storing the program counter and resuming execution of critical sections from exactly the same point where a crash occurred [21]. JUSTDO logging assumes caches are persistent, and has severe programming model restrictions such as volatile data cannot be used inside critical sections and compiler optimizations like register promotion have to be disabled. In fact, JUSTDO logging is 2-3× slower than Atlas on systems with volatile caches, which is the environment that NVthreads targets. Although NVthreads and Atlas both infer failure atomic regions from critical sections, NVthreads' approach is very different. NVthreads tracks data modifications at the granularity of virtual memory pages, isolates thread execution via forking processes, and uses redo logs instead of undo logs. Our evaluation shows that Atlas incurs high overheads of flushing data, and NVthreads outperforms Atlas by 2× on many applications. SoftWrAP uses cache-line combine of writes and asynchronous writes to logs to improve application performance [20]. Unlike SoftWrAP, NVthreads automatically infers failure atomic sections, uses operating system techniques to track updates, and is easy to integrate with `pthreads` based applications.

Table 1 summarizes the benefits of NVthreads' design decisions over existing persistent programming systems.

Operating system mechanisms. NVthreads' use of page-level tracking to provide crash tolerance is similar to RVM [30] and Rio-Vista [25]. Unlike RVM, NVthreads does not have the limitation that data needs to fit in memory, nor does it require DRAM to be battery backed as in Rio-Vista. Additionally, RVM and Rio-Vista don't infer consistency semantics from synchronization operations. QuickStore [36] and Texas [32] use virtual memory techniques to provide persistent object stores on disks, but are more appropriate for object oriented programs. DThreads [24] and Determinator [7] are systems that use multi-process execution to isolate threads for deterministic execution. NVthreads uses the DThreads library to manage data modifications in critical sections, but removes the approach of a global token that DThreads uses for deterministic execution, which can result in up to 9× application slowdown. NVthreads uses a per-mutex token mechanism that improves concurrency in some applications. Unlike DThreads, NVthreads also includes mechanisms to track dependence between critical sections, manage redo logs, and recover from crashes.

Issue	Commonly used solutions	NVthreads approach	NVthreads advantage
Determine consistency points	Use durable transactions [16, 34]	Infer from synchronization points	Ease of programming
Handle transaction aborts	Use undo logs [14, 26], word level STM [34]	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 [14] or word [34]	Intercept page writes	Amortized costs, good performance
Volatile caches	Flush writes, some require new hardware [16, 17, 21]	Flush log entries (memory pages)	Amortized costs, no processor changes

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

Whole system persistence advocates the use of residual power supply energy to flush data during a failure [27]. It relies on new hardware to provide the flush on failure feature, and its software for saving data during failure is susceptible to operating system crashes. Other transparent checkpoint-restore mechanisms rely on virtualization, which increases overheads for all applications or doesn't determine when checkpointing should occur so that data structures are consistent even in the presence of multi-threading [6, 22, 28].

Databases and transactions. Most databases use ARIES [26] write-ahead logging which was created to handle the performance difference between sequential and random disk accesses. MARS uses editable atomic writes to make NVM-specific choices such as eliminating undo logs [15]. Stasis also uses write-ahead logging and LSN-free pages to build durable data structures [31]. Similar to MARS, NVthreads does not need an undo log but it is because NVthreads buffers updates in process memory.

3. Programming model

NVthreads adds durability semantics to existing multi-threaded programs. We assume that both DRAM and NVM devices exist. NVM devices can be accessed by memory instructions as well as traditional filesystem interfaces. We consider processor caches to be volatile. Programs may control ordering of writes to NVM by using a combination of cache flush instructions, such as `clflush` or the upcoming optimized `clflushopt`, fences, and the `pcommit` instruction. We define crashes to be failures that result in the loss of all processor and DRAM state, but do not corrupt NVM state. Crashes can be caused by power failures or fail-stop software faults.

Original applications use locking primitives and multi-thread functionality provided by the `pthreads` programming model and library. Applications that link to the new NVthreads library become crash tolerant, and may need to incorporate recovery code to resume execution. We assume that programs modify shared persistent data within critical sections that demarcate failure atomic regions. Program data structures may be inconsistent within a critical section, but data structures are always consistent outside critical sections (when no locks are held).

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.

Guarantee. NVthreads guarantees that in the event of a crash failure, and after the completion of NVthreads' recovery process, application data structures will be in a consistent state in NVM, i.e., *the program state after recovery is as if the program stopped when no locks were held*, and the implementation guarantees it by providing the appearance of stopping at the exit of a critical section.

3.1 Persistent regions

A persistent region is a segment of memory, such as a memory mapped file, which is backed by non-volatile memory. NVthreads uses persistent regions to store application data durably. Data structures stored in persistent regions survive application crashes due to faults or power loss. Applications can allocate memory from persistent regions via `nvmalloc`, a persistent memory allocator. When allocating persistent objects, applications can install a handle, such as a variable name, that acts as an entry point to the object. After a crash, the recovery program may use the handle to determine the location of the persistent object and to traverse other reachable objects. For example, the recovery program may start from the head of a list, and traverse it to find elements in the linked list. Data not in persistent regions, such as application data in DRAM, are lost when a program terminates correctly or incorrectly, or if the system crashes. Applications can continue to allocate and access volatile memory using existing interfaces such as `malloc`.

3.2 Inferring consistent program points

A key challenge in adding durability semantics to programs is to determine when data structures are consistent. Instead of using durable transactions and rewriting applications, NVthreads infers failure atomic regions from synchronization primitives. NVthreads' API also supports programmers who explicitly specify commit points, similar to manual checkpointing. However, this paper focuses on how NVthreads can automatically infer failure atomic regions, thus making it easier for programmers to add durability semantics to their programs.

Multi-threaded applications use synchronization primitives, such as locks, to safely modify shared data structures. NVthreads uses these synchronization points as the boundary for failure atomic regions. We assume that programs are data race-free and data structures are consistent at synchronization points. However, updates inside critical sections can

```
1  //Well nested section
2  pthread_lock(&m1);
3      pthread_lock(&m2);
4      ...
5      pthread_unlock(&m2);
6      ...
7  pthread_unlock(&m1);
```

```
8  //Lock chaining
9  pthread_lock(&m1);
10     pthread_lock(&m2);
11     ...
12 pthread_unlock(&m1);
13     ...
14     pthread_unlock(&m2);
```

Figure 2: Different types of nested critical sections.

```
1  //Thread T1
2  a = 0;
3  pthread_lock(&m1);
4    pthread_lock(&m2);
5      a = 42;
6      pthread_wait(&cv, &m2);
7      ...
8    pthread_unlock(&m2);
9    //crash
10 pthread_unlock(&m1);
```

```
1  //Thread T2
2
3
4  ...
5  b = 0;
6  pthread_lock(&m2);
7    b = a;
8    pthread_signal(&cv);
9  pthread_unlock(&m2);
10 ...
```

Figure 3: Dependence between nested critical sections.

leave data structures inconsistent due to an untimely crash. NVthreads uses two rules to guarantee that data structures in persistent memory are always consistent: (1) if an application crashes while a thread is in a critical section, NVthreads ensures that updates to persistent data in the critical section are not visible in NVM, and (2) if an application crashes when a thread is outside of a critical section, NVthreads guarantees that only modifications till the last successfully executed critical section are made visible to NVM. The challenge for NVthreads is to ensure that these rules hold true even in the presence of multi-threading and volatile caches. When using NVthreads, the programmer does not need to instrument every synchronization call in the source code. This programming model is similar to that of Atlas [14]. Atlas relies on the compiler to detect and instrument failure atomic regions. NVthreads achieves the same effect by intercepting synchronization calls at runtime.

Returning to the example in Figure 1, if the program crashes at line 9, then the last element in the list will not have a back-pointer and the variable `tail` will no longer point to the last element. NVthreads avoids leaving the list in such an inconsistent state by making the critical section failure atomic. Let's assume that there are two threads `T1` and `T2` executing the critical section to append elements to the list. If `T1` has successfully completed the critical section and `T2` crashes in the middle of executing the critical section, then NVthreads will ensure that the persistent list, after program recovery, has only one new element.

Dependent critical sections. Locks lead to multiple kinds of critical sections. There may be cases with (1) an inner critical section completely surrounded by the outer critical section (perfect nesting), (2) two critical sections that overlap, such as lock chaining [13], or (3) critical sections that use condition variables. In all these cases, additional care is taken by NVthreads to ensure data structure consistency. Unlike Atlas [14], NVthreads supports condition variable operations for wait and signal calls without requiring them to be replaced with lock and unlock operations.

Figure 2 shows two examples of nested critical sections, one with perfect nesting and another with overlapping critical sections. Note that the case where critical sections don't nest perfectly does not arise in programs with transactions because transactions are scoped regions without partial overlap [16, 34]. NVthreads treats nested critical sections as a single failure atomic region. Logically, the system provides atomic durability to the smallest program region that encompasses all lock acquires and releases in a particular nested critical section. In Figure 2, NVthreads will guarantee that lines 2-7 (and 9-14) are atomically durable. For example, if a crash occurs in line 6 then changes made even in the internal critical section (lines 3-5) are undone. NVthreads *tracks dependence* between nested critical sections to correctly reflect updates in NVM. For example, the inner critical section in lines 3-5 is dependent on the outer critical section, and should become durable only after line 7 successfully completes.

Figure 3 shows another example of why NVthreads tracks dependence between nested critical sections. Line 3-10 in `T1` has nested locks and a condition variable, but it is a single failure atomic region, i.e., even if a crash occurs at line 9 the value of `a` should be 0 after recovery. However, in this example `T2` will set `b` to 42 (line 7 under `T2`) before the crash occurs. After recovery, this will lead to an inconsistent state where `a` is 0 but `b` is 42. NVthreads ensures that such cases do not arise by tracking dependence between critical sections. In this example, the critical section in `T2` is dependent on `T1`, and the changes in `T2` will not be visible in NVM unless `T1` completes the nested critical section. Section 4.1 describes how NVthreads implements dependence tracking.

3.3 Recovery code

In the event of a crash, NVthreads guarantees that program data structures are durable in NVM up to the point of the last successfully completed critical section. The recovery component consists of two parts. First, applications invoke NVthreads' recovery function, `nvrecover`, to apply log entries to NVM-resident data. This component is application agnostic. Second, similar to other systems, programmers may need to write application specific recovery code to resume execution after a crash [14, 34]. The user-provided recovery code is primarily used to assign NVM data to program variables, such as reading back two separately allocated arrays that are fields of a single variable, and assigning them to the appropriate fields.

While the amount of user-provided recovery code depends upon the complexity of an application, most programs simply need to call the application agnostic `nvrecover` for each allocated data and assign the output to program variables. In our experience, the recovery code for many machine learning and graph algorithms is only a few lines that read

```
1  /* points: input 2D points
2     labels[i]: id of center closest to point i
3     Only 'labels' is persistent  */
4  // Main iterations
5  float* kmeans(float* points, float* labels){
6     centers = calculateCenters(labels);
7     while (!converged){
8        pthread_create(.., findDistance, ..);
9        pthread_join(..);
10       centers = updateCenters(labels);
11    }
12    return centers;}
13 // Calculate distance for subset of points
14 void findDistance(points, labels, centers){
15    //find closest center for points with id in [X,Y]
16    pthread_lock(&m_xy);
17    labels[X:Y] = closestCenters[..];
18    pthread_unlock(&m_xy);
19    ...
20 }
21 //Recovery code
22 void main(){
23    if (crashed())
24       nvrecover(labels, N*2*sizeof(float),'labels');
25    else
26       labels=(float*)nvmalloc(N*sizeof(float),'labels');
27       ...
28    ans = kmeans(points, labels);
29 }
```

Figure 4: Pseudo-code for multi-threaded K-means.

core data structures from NVM and restart iterations to run until convergence (Figure 4). While Atlas [14] requires the programmer to check the crash status of each persistent region during recovery, NVthreads provides the programmer with the API call `crashed` to determine if all persistent regions need be recovered before normal execution.

3.4 Garbage collection

Due to crashes, applications using non-volatile memory have to handle cases of persistent memory leaks and dangling pointers. As an example, if a persistent object stores a pointer to volatile memory then after a crash it will point to garbage since volatile contents are lost. Such programs are discouraged, but NVthreads currently does not enforce these pointer restrictions. Similarly, after recovery if programs do not reuse persistent data that they store, the memory space will be wastage. NVthreads assumes that a garbage collector exists for the persistent regions (similar to file system checkers). This garbage collector should run after a crash, collecting unreachable memory, and flagging dangling pointers.

3.5 Example: K-means clustering

Iterative machine learning algorithms, such as clustering on large datasets, can take hours to converge even with multiple CPU cores (Section 5.5). Therefore, after a crash it is beneficial to restart the program from the last completed iteration instead of the beginning. Figure 4 is an implementation of K-means algorithm 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

Algorithm 1 Execution flow for each thread T

```
1:  while T has not terminated do
2:      w_T ← ∅                                   ▷ Initialize write set
3:      while instruction i is not a synchronization event do
4:          if i is a memory store then
5:              w_T ← w_T ∪ {page with memory address}
6:          end if
7:          Run i on priv. copy of global state   ▷ Avoids fine-grained logs
8:      end while
9:      log(w_T)                                  ▷ Log diff of modified pages
10:     Merge diff of pages from private copy to global state
11:     Execute synchronization event
12: end while
```

`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 5-12 the centers are first initialized using the current labels. In each iteration threads calculate the closest center to a subset of points, and update the corresponding labels in a critical section (lines 16-18). When a thread exits the critical section NVthreads guarantees that the labels of all the points it was working on have been updated. Lines 23-24 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 the `nvrecover` function 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`.

4. Design and implementation

Algorithm 1 shows how NVthreads implements its persistent programming model. As each thread executes, the write set tracks updates, which are initially performed on a process private copy of data. At synchronization points, modified data is first logged and then merged with the global state. For simplicity we have not shown the dependence tracking for nested critical sections.

An important challenge is to implement Algorithm 1 while ensuring good application performance. Unlike prior solutions, NVthreads uses operating system techniques to reduce application overheads. It leverages DThreads' approach [24] to execute a multi-threaded program as a multi-process program, though reducing the overheads imposed by deterministic execution. NVthreads also tracks dependence between critical sections, creates redo logs, and flushes log entries to NVM for recovery.

4.1 From threads to processes

NVthreads converts threads into child processes that execute in isolation until a synchronization point is reached. Typical synchronization points are lock acquires and releases, as well as thread creation and exit. The NVthreads runtime intercepts

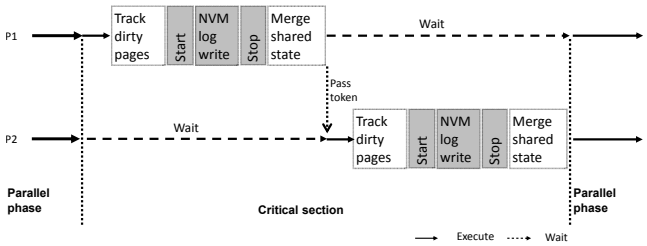


Figure 5: Overview of thread execution in NVthreads.

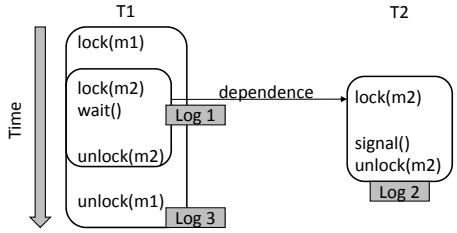


Figure 6: Tracking dependence between durable regions.

all synchronization calls to log and merge memory pages. At a synchronization point, changes made by child processes are applied to the original shared pages and made visible to all. This merge phase ensures that the behavior of the multi-process execution corresponds to a valid multi-threaded execution: threads can access and modify shared data.

Tokens. Since multi-threaded programs running under NVthreads become multi-process, we use the term token to denote mutexes visible across processes. Figure 5 shows an example of how execution proceeds in NVthreads. We assume that the application has two threads (`T1` and `T2`) that become processes when executing under NVthreads. This process-based isolation is used to buffer uncommitted writes. Outside the critical sections, processes execute in parallel. Inside a critical section, such as a locked region, processes execute sequentially, i.e., one after the other. Sequential execution is enforced by making each process wait for a per-lock token. This per-lock token is logically similar to a mutex in a multi-threaded program, except that it is visible across processes using shared memory. Only the process that acquires the token can enter the corresponding critical section. Others wait till the token is released. Once a process exits the critical section it writes its dirty pages to the log and flushes the log to NVM for durability. It then merges its modifications to the shared state, which makes the local updates visible to everyone, and finally passes the token to the next waiting process. It is worthwhile to point out that DThreads uses a single per-program global token to ensure there is a deterministic order of thread interleaving. This global token reduces concurrency: it serializes even those threads that access completely different mutexes. For deterministic ordering among threads, DThreads also forces all threads to wait at a barrier each time a mutex is released. This barrier can result in poor performance if there is load imbalance among threads. NVthreads reduces the overheads in some applications by admitting more thread interleavings (Section 5.3).

Copy-on-write. Child processes in NVthreads are created using the `clone` system call and each process gets a copy-on-write version of the program data. Shared memory regions are backed by a file, which is mapped into each process' address space and updated at synchronization points. Specifically, each process has two references of program memory: shared and process-local, which are created through two different `mmap` calls to the same file. Pages are initially read-only outside the critical section. Once processes write to pages, NVthreads' page fault handler uses `mprotect` on process-local pages with PROT_READ or PROT_WRITE and MAP_PRIVATE flags– effectively creating a copy-on-write page for local modifications. During the merge phase, the runtime compares dirty pages from a child process with the original versions of the shared pages, and applies the bytes modified by the child to the shared state. Only the dirty private bytes (i.e., diffs) are applied to the shared pages at synchronization points. At the end of a critical section, NVthreads releases the private copies and redirects references of these addresses to the shared pages with read-only permission set.

Dependent critical sections. The NVthreads runtime tracks dependence between durable regions to correctly handle nested critical sections and condition variables. NVthreads buffers the generated logs till all threads that are dependent on each other exit their critical section. Internally, NVthreads maintains metadata using hash tables to quickly locate the state of memory pages. The metadata is used to resolve data dependencies by performing hash table lookups on the page number. When the lock count of the dependent critical section stored in the metadata drops to zero, NVthreads marks the memory pages as ready and then resets the metadata when exiting the outermost lock. Figure 6 shows how NVthreads tracks this dependence. If a thread `T1` is inside a nested critical section, and passes a token to thread `T2` for execution, then `T2` is dependent on `T1` if `T2` touches a page that `T1` modified in the current nested section. Only when `T1` completely exits its nested critical region are the logs of `T1` and `T2` together committed to NVM. In Figure 6, logs 1, 2, and 3 are written to NVM only when `T1` unlocks `m1`.

4.2 Logging

Figure 7 illustrates how logging works in NVthreads. Since NVthreads uses multi-process execution, data structure modifications are initially available only in the private copy used by each process. Only at synchronization points, and after dirty pages have been merged with the shared program state, will the changes be potentially reflected in NVM (depending upon when cache lines are evicted). NVthreads requires only logs to be durably flushed from caches to non-volatile memory. Except for the log truncation operation, which we describe later, NVthreads' correctness guarantees do not depend upon

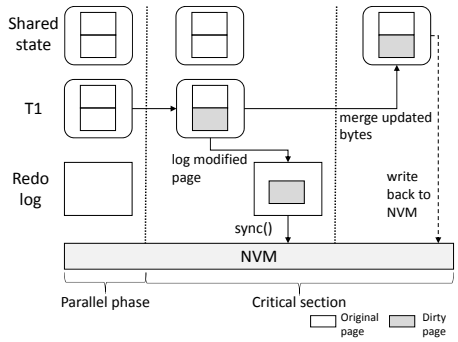


Figure 7: Overview of logging.

```
1  //Fine-grained.
2  pthread_lock(&m1);
3   l = log(&a)
4   sync(&l);
5   a = v1;
6   sync(&a);
7   l = log(&b);
8   sync(&l);
9   b = v2;
10  sync(&b);
11 pthread_unlock(&m1);
```

```
1  //Coarse-grained(NVthreads).
2  //Writes are initially to
3  //a private copy of data.
4  pthread_lock(&m1);
5   a = v1;
6   b = v2;
7   ...
8  pthread_unlock(&m1);
9  log_pages();
10 sync();
11 merge_pages();
```

Figure 8: Fine-grained [14] vs coarse-grained tracking.

when application data is flushed from caches, which reduces the overhead of eagerly flushing cache lines.

Ordered writes to NVM. Since NVM devices appear as memory, the processor may cache data and reorder updates. NVthreads has to ensure that applications can correctly recover even in the presence of volatile caches and re-ordered writes. NVthreads requires two mechanisms from the hardware, which are available in most processors, and are also expected to be present in NVM systems. First, NVthreads requires a way to evict cache lines to NVM, such as the `clflush` or `clflushopt` instruction. NVthreads uses these calls to keep its log consistent and durable. Second, NVthreads assumes that the system provides fences, such as `mfence`, to ensure that instructions prior to a fence complete before those after the fence. Applications should also be able to use operating system functionality, such as `msync` or `fdatasync`, to write out logs to NVM. The `fdatasync` call, when used as a blocking call, ensures that all writes have safely reached the device before returning.

Coarse-grained memory management. Figure 8 contrasts fine-grained memory management [14, 34] with NVthreads. In fine-grained memory management, first each NVM write has to be intercepted in software, then log entries are made durable followed by program data. The `sync` operator ensures writes reach the NVM and are ordered. It involves draining the volatile cache line using `clflush` followed by a barrier for instruction ordering. In contrast, NVthreads uses process local pages to buffer writes, and at the end of the critical section, logs modified pages. This approach allows NVthreads to avoid intercepting each NVM write. Additionally, only writes to the log need to reliably reach the NVM. The log provides the opportunity to batch pages and use the unordered but faster `clflushopt` instruction instead of `clflush`.

Redo log. NVthreads uses a redo log and stores the log in an NVM file system (Section 5.1) for crash recovery. Data structure modifications are first committed to the log and then made durable in the application's working space (Figure 7). NVthreads uses the `mprotect` system call and a custom page fault handler to track dirty pages. Right before dirty pages of a process are merged with the shared program state, the dirty portions of the pages (i.e., diffs) are written out to the log. For correctness, the log has to be made durable before merging changes with shared program state. Otherwise, application data may reach the NVM before the log, and a crash in between will result in a case where the system cannot undo inconsistent data structures in NVM. NVthreads uses `fdatasync` to write out log entries to NVM. After writing out the log entry, NVthreads also writes out a special end-of-log symbol. This symbol is used during recovery to identify if all of the items in a log append were written to NVM. The end-of-log symbol is durably written before the current process continues to merge its dirty pages to the shared program state.

Unlike the log, NVthreads does not force application data to be flushed from volatile caches after each merge phase, thus reducing cache flush overheads. The reason is because the log has sufficient information to recover data in the case of a crash. This approach of not forcing data out to durable media is similar to ARIES like protocols, i.e. *no-force* in database parlance [26].

No undo log. NVthreads does not create undo logs. In the ARIES style of database logging, the modified data of in-flight transactions may be paged out to the disk to make space in the buffer manager. In Atlas [14] updates are made in-place. Therefore, these systems need an undo log to remove the effects of uncommitted transactions or durable sections. In NVthreads each process makes its modifications on a private copy-on-write version of data, and the undo log is unnecessary.

Log truncation. Since reapplying the log from the beginning can be slow, NVthreads supports log truncation to reduce the number of log entries that need to be replayed during recovery. The redo log can be truncated up to a particular entry if the system can guarantee that all changes up to that entry are reflected in NVM. Since the application writes its data directly to NVM, the only issue is to ensure that data in the volatile caches makes it to NVM before the corresponding write entries in the log are truncated. In NVthreads, log truncation is triggered periodically in the merge phase of the synchronization points, when NVthreads has control over the whole program, and the application is quiescent. NVthreads first flushes all cache entries, and then truncates the log, using a fence to order the cache flush before log truncation.

4.3 Recovery

Applications initiate the recovery process if they are restarting after a crash. NVthreads logs the modified portion of each page, i.e., the diff, before they are applied to the shared state at a synchronization point. During recovery, all the log entries are applied to recreate application data pages.

Applications invoke `nvrecover`, the NVthreads recovery function, which first replays the redo log, applying entries in the log to corresponding pages in the NVM (Figure 4). Once the recovery process is complete, it is guaranteed that the application's data structures in the NVM are consistent as of the last completed critical section. Some applications may have additional user written recovery code to read data structures from the NVM, assign them to program variables, and resume execution.

5. Evaluation

We evaluate NVthreads to answer the following questions: (1) What are the overheads of making programs durable? (2) What are the benefits of NVM over flash storage? (3) How important is fast recovery? and (4) How does NVthreads compare to Mnemosyne and Atlas?

5.1 Setup

All experiments were run 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.

Applications. We used 14 multi-threaded benchmarks from PARSEC [11] and Phoenix [29], and ran them with the configuration of a *large* dataset. We also evaluate NVthreads on the PageRank graph algorithm, K-means clustering, and Tokyo Cabinet, which is a B+-tree based key-value store. We did not enable log truncation, as the logs easily fit the main memory of our server.

NVM emulator. Since NVM devices are commercially unavailable, we use a simple emulator [1] to measure the effect of different NVM latencies on performance. As NVthreads relies on a NVM filesystem to store its log, the emulator consists of a modified Linux `tmpfs` on DRAM in which software delays are injected to each read and write filesystem call. These delays are created by reading the processor timestamp via `RDTSCP` instruction and spinning in a loop until the counter reaches a certain value. Since flushing a 64B cache line with the `clflush` instruction costs around 100ns, the cost of flushing a 4KB memory page is around 6,400ns. According to Intel [5], the mircroarchitecture could improve flushing overhead by 8× by using the pipelined `clflushopt` instruction that optimizes flushing large buffers. Therefore, in all experiments, we add 1,000ns delay to each 4KB page write, which models the expected overhead for write barriers and cache flushes with `clflushopt` [19]. In Section 5.4 we vary the page write delay to measure the impact of higher cache flush overheads.

[1] https://github.com/HewlettPackard/dummy_nvmfs

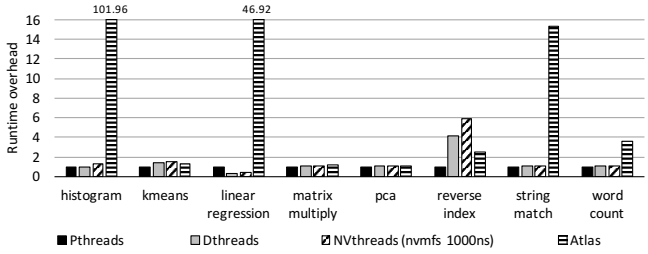
5.2 Porting effort

Instead of modifying the applications in PARSEC and Phoenix to call `nvmalloc` and persist only a select set of data structures, we made all heap allocated data persistent by ensuring each call to `malloc` is a persistent allocation. This setup ensures that we measure the worst case overhead, i.e., when all data is durable, without modifying the application at all. We did not add restart code to applications for this experiment, since our goal is to observe overheads during normal execution. To study the benefits of resuming a program after a crash, we made K-means crash-tolerant by modifying four lines in the program, which `nvmalloc` and `nvrecover` the labels array. For Tokyo Cabinet, we set the `BDBOTSYNC` flag in the vanilla benchmark program when opening a database file. This flag ensures that every transaction synchronizes its modified contents with the storage device. Since NVthreads automatically checkpoints progress after every transaction, we made all data durable and removed the `BDBOTSYNC` flag when using Tokyo Cabinet with NVthreads.

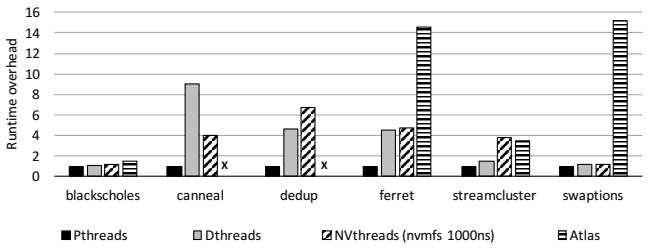
5.3 Performance

Figure 9 shows the overheads of using NVthreads on all 14 applications from the Phoenix and PARSEC benchmarks. In Figure 9 each program uses twelve threads. The baseline is programs running in DRAM with `pthreads`. The DThreads performance numbers give a sense of the overheads of converting multi-threaded execution into multi-process execution. NVthreads numbers are on the modified `tmpfs` with a 1,000ns delay to each page write. We also show the performance of these applications when made durable using Atlas, which uses fine-grained cache line flush based logging.

Our results show that for 9 out of 14 applications, NVthreads makes the application durable with less than 28% overhead. The application `linear regression` has better performance with NVthreads than with `pthreads`, because multi-process execution reduces false sharing. Only `reverse index`, `canneal`, `dedup`, and `ferret` have more than 4× overhead compared to unmodified applications using `pthreads`. The DThreads bars in Figure 9 show that these programs incur considerable overhead when using multi-process execution. However, NVthreads' can improve concurrency in some applications where DThreads would have serialized thread execution due to the global token. As an example, even though `canneal` with NVthreads is 4× slower than the `pthreads` version, it is actually 2× faster than the non-persistent DThreads version. Similarly, `reverse index` would have been 7× slower, instead of 5.5×, if the global token approach was used. At the time of writing this paper, only `reverse index` and `canneal` use the per-mutex token version of NVthreads, and we ex-



(a) Phoenix applications



(b) PARSEC applications

Figure 9: Cost of crash tolerance. Lower is better.

Table 2: Application characteristics.

Program	Critical sections	Logging (%tot. time)	Logged pages	Log size	Slowdown over pthread
histogram	25	12.56%	44	0.3MB	1.2
kmeans	1845	20.91%	9763	46MB	1.5
linear reg.	25	12.87%	27	0.2MB	0.3
matrix mult.	37	2.32%	3955	16MB	1.1
pca	25	18.11%	11463	45MB	1.1
reverse index	137113	37.88%	2691474	11GB	5.5
string match	37	3.24%	39	0.3MB	1.1
word count	145	1.72%	12476	50MB	1.1
blackscholes	25	6.76%	89	39MB	1.2
canneal	1475	39.34%	7440183	29GB	4.1
dedup	320492	33.72%	2314600	11GB	6.7
ferret	8010	4.93%	149963	618MB	4.8
streamcluster	95581	47.07%	176054	1.1GB	3.7
swaptions	25	4.76%	483	2.0MB	1.2

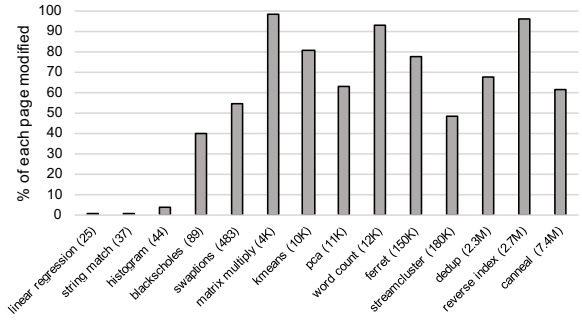


Figure 10: Average percentage of each 4KB page modified by an application. Applications are ordered by total number of modified pages, which is shown in brackets.

pect the overheads of other applications to decrease when integrated with the per-mutex token version of NVthreads.

Figure 9 shows that NVthreads performs as well, and in most cases significantly better, than Atlas. We were not able to get Atlas to run on the two challenging workloads of `canneal` and `dedup` in the PARSEC benchmark. In 10 out of the 12 workloads on which Atlas ran, NVthreads is from 7% to 100× faster depending upon the workload. NVthreads is about 7% and 50% slower than Atlas for `streamcluster` and `reverse index` respectively. In `reverse index`, most of the overheads in NVthreads are due to the multi-process execution as shown by the DThreads bar. Section 5.6 has more in-depth comparison with Atlas.

Application characteristics. Table 2 shows the characteristics of all benchmarks by measuring the number of critical section invocations and logging overheads. The Phoenix benchmarks primarily include data mining applications. Except for `reverse index`, NVthreads has to log only 27 to 12K dirty pages in Phoenix applications, and results in only 20% slowdown over the `pthreads` version. In comparison, for some of the PARSEC applications such as `dedup`, NVthreads has to manage and log about 2.3 million dirty pages and spends 34% of the total time in logging. This high logging overhead, in addition to multi-process execution, is a reason for `dedup`'s 6.7× slowdown.

Our experiments also reveal that NVthreads has low memory footprint and uses at most 400MB more DRAM space versus the `pthreads` version.

Page-level tracking. Figure 10 shows what percentage of each page is modified by the application. It can help us understand whether page-based tracking is a good option compared to byte level tracking. There are two interesting observations. First, 9 out of the 14 applications modify more than 55% of each page. This means if these applications modify a page, they generally write more than 2KB of data to the page. This makes it worthwhile to track data at the granularity of a page. Second, of the 5 remaining applications that write only a few bytes per page, 4 of them (`linear regression`, `string match`, `histogram`, and `blackscholes`) modify fewer than 90 pages during the execution of the program. These applications have few overall writes and, as shown in Figure 9, they are less than 20% slower on NVM with NVthreads than running with `pthreads` on DRAM.

Scalability. We also evaluated whether NVthreads programs scale similar to their `pthreads` counterparts as core count increases from 1 to 12. Our results reveal that `pthreads` provides low to moderate speedup for the 14 applications, never reaching more than 6.5× speedup over a single core. When using NVthreads 10 applications show similar scalability as with `pthreads`. Of the remaining four applications,

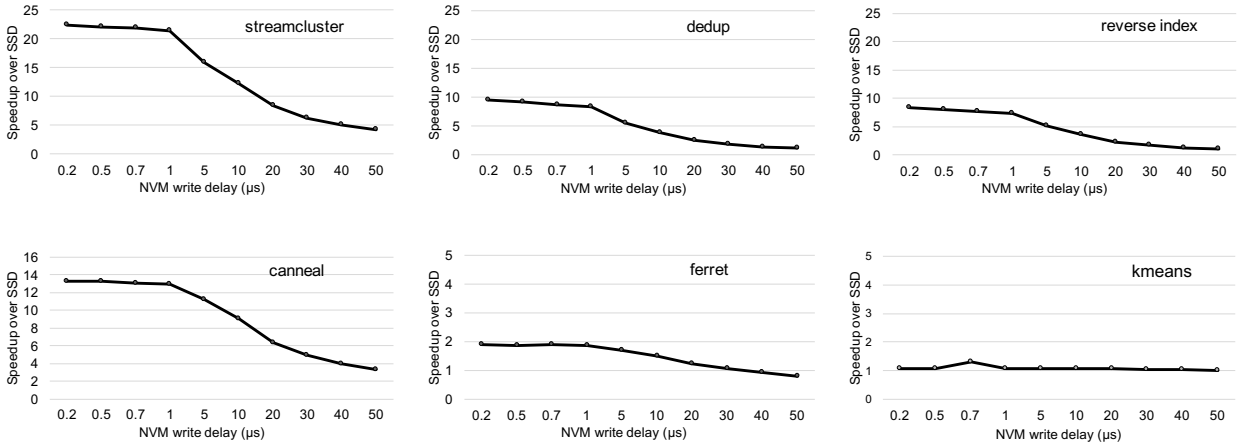


Figure 11: Effect of NVM page write delay on application performance. Speedups are over SSD. Higher is better.

`ferret` and `dedup` with NVthreads scale about 70% less than `pthreads`. In `stream cluster`, the `pthreads` version with 12 cores is 3× faster than one core, but the NVthreads version is only 1.3× faster than single core. Finally, for `canneal`, the `pthreads` version at 12 cores is merely 1.3× faster than one core, while the NVthreads version does not scale at all. The low scalability of these 4 NVthreads applications is because of the high synchronization costs and write counts (Table 2).

5.4 Benefits of using NVM versus SSDs

We measure the speedup of programs on NVM over the same programs using `ext4` on a solid state drive (SSD) to store the logs. We vary the 4KB page write delay to NVM from 200ns (DRAM-like) to 50μs (Flash-like).

Latency-sensitive applications. Figure 11 shows five applications whose performance improves substantially with NVM: `streamcluster`, `dedup`, `reverse index`, `canneal`, and `ferret`. There are two interesting observations. First, these applications can be 2× to 22× faster on NVM compared to using SSDs. For example, `streamcluster` is 22× faster on low-latency NVM than SSD. Second, performance of these applications drops only if NVM page write latency is much more than 1μs. These applications track and write many dirty pages, and will benefit from NVM hardware as long as NVM devices are reasonably faster than SSDs.

Storage-agnostic applications. The remaining nine applications show little difference as we vary NVM latency. In Figure 11 we plot the performance of only `kmeans`, which is representative of all nine storage-agnostic applications. Although `kmeans` spends 20% of the execution time to log memory pages, the total size of the log is only 46MB, which is relatively small compared to the log size generated by latency-sensitive applications. Storage-agnostic applications have around 15% performance benefit when using NVM as compared to using an SSD. Note that these applications were anyway incurring a mere 28% overhead compared to their original `pthreads` versions. They depict a spectrum of applications where NVthreads incurs very little logging overhead to add durability, and we expect programmers to run them with NVthreads even on today's storage systems.

5.5 Benefits of recovery

Figure 12 shows the benefits of adding durability to programs. We ran K-means and introduced a crash during its execution. The input data (1M, 20M, and 30M 3-D points) have to be clustered into 1,000 groups. On our datasets, K-means converges after approximately 155 iterations.

In Figure 12, the x-axis shows the iteration when the crash occurred. We plot the speedup compared to K-means with `pthreads`, i.e., the program has to restart from the beginning in the event of a crash. Under these circumstances the maximum possible speedup via recovery on any program is 2×, which happens when the program crashes just before completion, thereby requiring everything to be redone. Figure 12 shows that for large inputs, if the crash occurs after 75 iterations, NVthreads' version of K-means converges almost 1.4× to 1.9× faster than starting from the beginning. As an example, for the 30M dataset, the NVthreads version converges 30 minutes earlier than the pthreads version. If the crash occurs too early in the computation, very little work is wasted, and recovery does not provide much benefit. For the small 1M dataset, although the time to converge is only a couple of minutes, NVthreads recovery is still useful if the crash occurs around iteration 150.

5.6 Mnemosyne and Atlas

We compare NVthreads with Mnemosyne [34] and Atlas [14] that use word or store level data tracking. We limit our evalu-

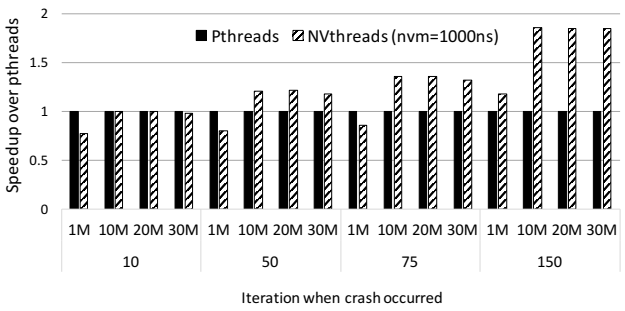


Figure 12: K-means recovery. Higher is better.

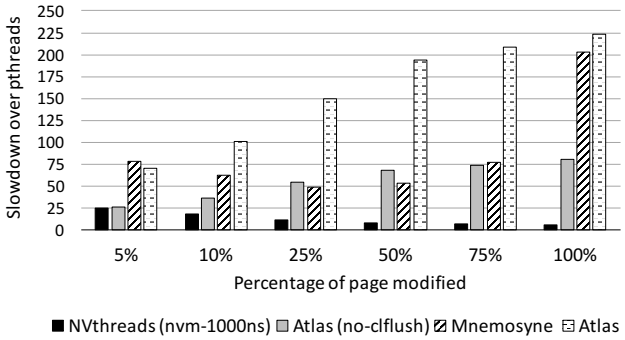


Figure 13: Benefits of NVthreads. Lower is better.

ation to microbenchmarks and a real-life graph application because it was challenging to get Mnemosyne to work with other benchmarks.

Microbenchmark. Mnemosyne crashes when we allocate more than 4MB of durable data. Therefore, we use a microbenchmark that allocates 1000 memory pages (4MB of data), and then 4 threads modify a random region of each page. In all cases, threads modify different pages so that the STM in Mnemosyne does not incur any concurrency control related aborts. Mnemosyne incurs a constant overhead of 2 seconds when forking threads and using `pmalloc`. We have excluded this overhead for Mnemosyne. Figure 13 shows the slowdown of each system compared to `pthreads`. Both Mnemosyne and Atlas are 70× slower than `pthreads` in most cases and more than 200× slower when threads modify 100% of each page. NVthreads incurs an overhead of 25× when only 5% of the data is modified, which decreases to 5× when each thread modifies 100% of each page. Since NVthreads tracks data at page granularity its overheads get amortized as a bigger fraction of each page is modified. As a result, NVthreads is 3×-30× faster than these systems. The bar *Atlas (no-clflush)*, shows that half of Atlas' overheads are due to cache flushes, which point to the high overheads of micromanaging NVM writes. Finally, without any log truncation, NVthreads uses 165MB to store metadata and logs, compared to 550MB-3.2GB by Mnemosyne and 70MB-1.4GB for Atlas.

PageRank. The previous microbenchmark magnifies the overheads of durability since the threads only perform persistent writes. Therefore, we also compare these systems on a real application, PageRank, which is an iterative algorithm to determine the importance of nodes in a Web graph [12]. First, we use the real-world *Slashdot* graph dataset which has 82K vertices and 950K edges [1]. We picked this graph because it is the largest input on which we could run Mnemosyne. Even on this small graph, NVthreads is more than 2× faster than Atlas and 10× faster than Mnemosyne at 12 cores, completing an iteration in 170ms compared to 500ms for Atlas and 5 seconds for Mnemosyne. After 10 iterations, NVthreads needs only 273MB space for logs compared to 580MB for Mnemosyne and 600MB for Atlas.

Unlike Mnemosyne, we were able to run NVthreads and Atlas on much larger graphs such as the 1.2 GB *Livejournal* data [8]. 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. NVthreads uses 300MB of log space per-iteration versus Atlas' 650MB.

5.7 Key-value store

Tokyo Cabinet is an high performance, open source library for database management [2]. It stores records as key value pairs. We configured Tokyo Cabinet to organize its records using a B+ tree. Tokyo Cabinet uses a memory mapped file to back data and periodically flushes data using `msync` to guarantee durability.

Our goal is to make the B+-tree in Tokyo Cabinet durable by linking it with NVthreads. Since Tokyo Cabinet and NVthreads both use page-based writes, we expect unmodified Tokyo Cabinet to have better performance since it uses custom code to implement transations for key-value stores. This experiment shows that NVthreads can be integrated without making any changes to the B+-tree. NVthreads incurs higher overheads but avoids the code complexity of a custom transaction system.

In Figure 14 we compare Tokyo Cabinet running on (1) an SSD (*TC-SSD*), (2) on our NVM emulator with injected delays (*TC-nvmfs*), and (3) when NVthreads is used to make the B+ tree durable on the NVM emulator (*TC-NVthreads*). In our workload we vary the key and value sizes from 64B to 4096B. We use 8 threads and perform 100,000 write operations. Figure 14 shows that when Tokyo Cabinet is used with an SSD (TC-SSD), its throughput varies from 1,300 updates/sec with 64B keys to 1,200 updates/sec with 4096B keys. When we use NVthreads to make the B+ tree in Tokyo Cabinet durable, its throughput on the NVM emulator ranges from 12,000 updates/sec for 64B keys to 5,000 updates/sec for 4096B keys, which is 4×-

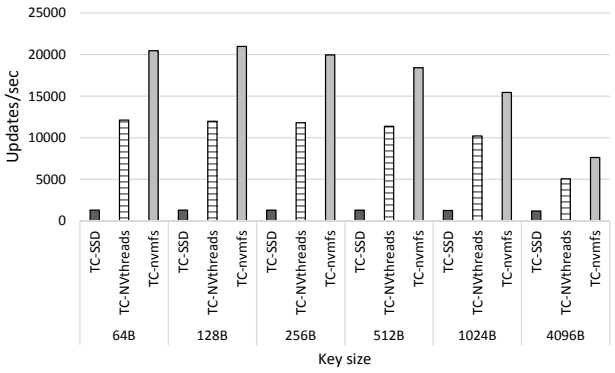


Figure 14: Throughput of Tokyo Cabinet. Higher is better.

9× better than using unmodified Tokyo Cabinet on SSDs. Tokyo Cabinet running on our NVM emulator achieves about 20,000 updates/sec with 64B keys and 7,500 updates/sec with 4096B keys. These numbers are about 30%-50% better than what one gets when using NVthreads. Even though the performance when using NVthreads is lower, it provides us an easy way to make the B+ tree durable by merely linking with NVthreads and without developing a custom key-value transaction system.

6. Discussion

Lock-free data structures. Programs using lock-free data structures are not supported in NVthreads. NVthreads intercepts synchronization points, especially lock acquires and releases, to checkpoint the execution state. Although it is possible for NVthreads to build function wrappers around atomic instructions, the NVthreads library would require non-trivial code refactoring, which may impact performance of lock-free data structures. We believe that lock-free data structures should be handled using orthogonal techniques to achieve good performance and failure atomicity.

Multi-threading libraries. NVthreads is a drop-in replacement for the `pthreads` library. Most Unix-based multi-threading libraries internally use the `pthreads` API. Therefore, NVthreads can intercept synchronization calls performed by these libraries and log memory pages. However, NVthreads cannot replace multi-threading libraries that do not rely on the `pthreads` API.

Memory overhead. The redo log in NVthreads is a key source of memory overhead. The size of the redo log depends on application characteristics: the more pages an application modifies during the execution, the more memory space NVthreads needs for storing the redo log. NVthreads also requires memory space for maintaining dirty pages during execution. NVthreads discards the old working copies of process-private pages at the end of a critical section. Thus, the increase in memory utilization for these pages is transient. The memory overhead of NVthreads is very low for applications that primarily read data.

We expect NVM technologies to provide persistence and higher capacity than DRAM, but at a fraction of the cost. Therefore, memory usage of NVthreads may become less of an issue in future NVM based systems.

Fine-grained vs. coarse-grained tracking. The main overhead in guaranteeing failure atomicity is due to the mechanisms for tracking and logging writes to data structures. Fine-grained logging techniques micromanage NVM writes and introduce non-negligible slowdown. NVthreads uses coarse-grained logging that amortizes logging overhead and outperforms fine-grained logging mechanisms on many applications. However, coarse-grained logging in NVthreads increases transient memory usage due to copy-on-write pages. Although both fine-grained and coarse-grained logging techniques require extra space for storing logs, our evaluation shows that NVthreads uses less space for metadata and logs, compared to fine-grained logging systems such as Atlas and Mnemosyne.

7. Conclusion

NVthreads uses fast non-volatile memory 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. Due to these techniques, NVthreads has good performance. Compared to state-of-the-art persistent systems, NVthreads significantly reduces the performance gap between unmodified applications and their crash tolerant versions.

While NVthreads advocates ease of use and the approach of coarse grained tracking, it has certain limitations. NVthreads piggybacks on synchronization primitives to demarcate failure atomic sections. Other explicit or hybrid approaches for detecting failure atomic sections will make the programming model more general. We expect NVthreads to co-exist with systems that embrace fine-grained tracking and logging. NVthreads may perform better on workloads with large amounts of writes in a page, but for certain workloads fine-grained tracking systems may be the appropriate implementation. There are multiple ways to improve our current prototype as well, by adding a memory manager to clean up memory leaks in persistent regions, and by using a combination of regular and huge pages to reduce new overheads seen in in-memory applications [10]. 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 non-volatile memory era.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Haibo Chen, for their invaluable feedback. We also thank Haris Volos, Dhruva Chakrabarti, and Hideaki Kimura for assisting us in evaluating NVthreads. This work was supported by Hewlett Packard Labs, NSF TC-1117065, and NSF TWC-1421910. P. Eugster was partly supported by European Research Council under grant FP7-617805.

References

[1] Stanford network analysis package. http://snap.stanford.edu/snap, 2009.

[2] Tokyo Cabinet: a modern implementation of DBM. http://fallabs.com/tokyocabinet/, 2010.

[3] Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/docs/DOC-6713, 2015.

[4] SanDisk and HP launch partnership to create memory-driven computing solutions. http://www8.hp.com/us/en/hp-news/press-release.html?id=2099577, 2015.

[5] 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, 2016.

[6] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[7] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[8] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[9] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[10] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[11] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

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

[13] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[15] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[16] 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. *SIGPLAN Not.*, 46(3):105–118, Mar. 2011.

[17] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[18] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. In *International Symposium on VLSI Technology, Systems and Applications*, VLSI '08, pages 70–71. IEEE, 2008.

[19] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[20] E. Giles, K. Doshi, and P. J. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *MSST*, pages 1–14. IEEE Computer Society, 2015.

[21] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[22] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 25:1–25:14, Berkeley, CA, USA, 2007. USENIX Association.

[23] 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*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[24] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[25] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM.

[26] 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):94–162, Mar. 1992.

[27] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[28] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[30] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, Feb. 1994.

[31] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[32] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: Good, fast, cheap persistence for c++. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 145–147, New York, NY, USA, 1992. ACM.

[33] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[34] 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, pages 91–104, New York, NY, USA, 2011. ACM.

[35] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 148–173, Berlin, Heidelberg, 2006. Springer-Verlag.

[36] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. *The VLDB Journal*, 4(4):629–673, Oct. 1995.

[37] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–10, 2015.