Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments

Douglas Santry and Kaladhar Voruganti, *NetApp, Inc.*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/santry

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments

Douglas Santry, Kaladhar Voruganti
NetApp, Inc.

Abstract

In this paper we describe a storage system called *Violet* that efficiently marries fine-grained host side data management with capacity optimized backend disk systems. Currently, for efficiency reasons, real-time analytics applications are forced to map their in-memory graph like data structures on to columnar databases or other intermediate disk friendly data structures when they are persisting these data structures to protect them from node failures. *Violet* provides efficient fine-grained end-to-end data management functionality that obviates the need to perform this intermediate mapping. *Violet* presents the following two key innovations that allow us to efficiently do this mapping between the fine-grained host side data structures and capacity optimized backend disk system: 1) efficient identification of updates on the host that leverages hardware in-memory transaction mechanisms and 2) efficient streaming of fine-grained updates on to a disk using a new data structure called Fibonacci Array.

1. Introduction

Increasingly organizations are finding a lot of business value in performing analytics on the data that is generated by both machines and humans. Not only are more types of data being analyzed by analytics frameworks, but increasingly people are also expecting real-time responses to their analytic queries. Fraud detection systems, enterprise supply chain management systems, mobile location based service systems, and multi-player gaming systems are some applications that want real-time analytics capabilities [10]. In these systems both transaction management and analytics related query processing are performed on the same copy of data. These applications have very large working sets, and they generate millions of transactions per second. In most cases these applications cannot tolerate network and disk latencies, and thus, they are employing main memory architectures [11, 13, 14, 15] on the host application server side to fit the entire working set in memory.

Even though these applications want to store the entire working set in main memory, for protection from node failures, they still need to persist a copy of their data off the application server box. Typically, copies are stored on disk/NAND flash based storage systems because these technologies are much cheaper than main memory. Thus, there is a bifurcation of IOPs optimized data management at the host and capacity optimized data management at the backend disk based storage system.

The key insight that is prompting the work in this paper is that there is a mismatch in the fine-grained data management model on the host and the block optimized data management model in the backend disk/flash based systems. For decades applications and middleware developers have been forced to map their in-memory fine-grained data structures onto intermediate block I/O friendly data structures. Despite the application running entirely in DRAM, the data structures that in-memory databases employ are little changed from when they lived on disks owing to the difficulty in persisting them to a block oriented device. This difficulty is retarding the development of in-memory systems. For ease of implementation the in-memory data structures are part of memory pages that are, in turn, mapped to disk blocks using data structures like B-Trees. However, fundamentally, there are the following inefficiencies in this approach and going forward these have to be resolved in order to provide an efficient end-to-end data management solution for the new emerging real-time analytics applications.

Problem Description

In the past, data structures have been designed to localize updates to a block in order to minimize random I/Os to a disk-based storage system. For example, the inventors of columnar databases observed that if an entire dataset has to be scanned, but only a subset of the columns are important, then a vertical decomposition of a database realized the streaming bandwidth of DRAM and disk subsystems. However, going forward, new types of main-memory graph data structures are emerging, such as *Voronoi Diagrams* [1], that are unconcerned about localizing their updates to a few blocks. These data structures can perform important analytic operations in $O(N)$ time where as a columnar database would require $O(N^2)$ time. *Voronoi Diagrams* are being leveraged in biology, chemistry, finance, archaeology, and business analytics domains. Similarly, middleware software is being designed to support these new emerging graph data structures [10, 17, 20].

When backing up the host side in-memory data structures on to a block-based storage system, it is desirable to be able to efficiently detect and transfer only the updated bytes of data instead of transferring the entire page on which they reside. As is shown in the experiment section of this paper, there are performance benefits in the end-to-end throughput by handling graph data structures natively (the focus of this paper) versus mapping them on to a columnar database or other intermediate data structures [17].

Contributions

In this paper we present a storage system called *Violet* that efficiently marries fine-grained host side data management with back-end block level storage systems. Violet divorces the problem of data structure selection and implementation, which should be wholly based on asymptotic requirements, from data structure persistence, which should not be the application writer's problem. The key highlights of this architecture are:

- **End-End Data Management Stack:** *Violet* presents a byte-oriented storage system that provides an integrated end-to-end storage stack that 1) efficiently detects fine-grained updates on the host 2) replicates the updates remotely on to a back end block based storage system and 3) efficiently streams the updates to a disk drive. The violet storage system architecture has both a host-side footprint and a backend capacity layer footprint.
- **Efficient identification of fine-grained updates:** Violet leverages a hardware transactional memory CPU instruction set (e.g. TSX instruction set from Intel) to detect the read/write changes to data within an in-memory transaction boundary. This allows Violet to track changes at a very fine-grained level in a multi-core CPU environment, and in turn, transfer changes (not the entire data block) off the node in an efficient manner. Violet also allows for both fine-grained data structure level snapshots at the host and coarser grained file level snapshots at the backend capacity optimized storage system.
- **Efficient streaming of sparse random I/Os on to disks:** Violet proposes a new data structure called the *Fibonacci Array* that enhances standard log structured merge trees [12] and COLA [2] data structure notions by leveraging the key insight that in the emerging main memory middleware/application architectures, the backend disk based systems primarily handle write operations as reads are mostly satisfied at the host servers.

The net result of the above innovations is that we are able to efficiently 1) detect 2) transmit and 3) persist fine-grained updates at the host to a block based backend storage system. Thus, emerging real-time applications and databases that are providing support for graph like data structures can leverage the benefits of the techniques being presented in this paper when designing their logging and off-node replication mechanisms.

2. Architecture

As shown in Figure 1, Violet is a distributed system that works as a cluster of cooperating machines. Violet is comprised of 1) a user-level library that gets linked with applications, 2) Violet servers that run on machines with disks attached (called *Sponges*) and 3) a master server that is in charge of the cluster. Applications manage memory with the user space library. The memory region is called a file; when storage class memory (SCM) becomes available Violet can mmap(2) a SCM file and the applications remain unchanged.

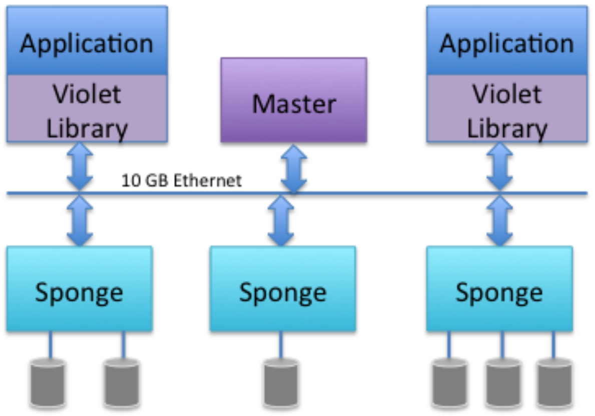

Figure 1: *The Violet Architecture.*

The era of SCMs has started with many vendors announcing the availability of different types of SCMs like PCM, ReRAM, and STT-MRAM [5].

The *primary* copy of a file is on the host in DRAM where the application runs. The copy on the host is a region of volatile anonymous pages created with mmap; so it is not a file on the host per se but this will change with the availability of SCM.

Sponges store the data on their locally attached disks. The set of Sponges that store a file on disk is called the file's replication group. The copies are used to provide disaster recovery and data management. Note that the Sponges only contain a *redundant* copy of a file on their disks. Violet is *not* a distributed memory system. The only updates to the file take place in the application's address space on the host machine. A file can only be written by a single application (the single-writer model is used by most commercial in-memory DBs).

Disks are much cheaper than DRAM, and are expected to be much cheaper than SCM, so basing node failure recovery on disks makes economic sense. As will be seen, a disk based failure recovery strategy does increase restore time (and hence downtime in the event of failure). An alternative approach would be to simply mirror in DRAM on a second host. This would increase the cost of the system accordingly, but it would increase availability as there would be a `hot' host to restart an application on in the event that its machine fails. The system presented here can support both the model where the second copy is on a peer, with the recovery copy on the backend disk, and also the model where the second copy is in DRAM.

3. Host-Side Client Library

This section describes the details of the host-side user space library. We present a C++ API that applications use to describe transactions, and the Violet library that must be linked. The Violet transaction machinery and how it replicates the memory updates to remote storage controllers is described.

The User-Land Storage Stack

The goal of Violet software is to provide disaster recovery and transaction support for applications that desire byte-oriented data structures. The most efficient place to do this is in user-land inside the application. While it is common to persist updates to a mmap'ed file with a page daemon running in the kernel, this would be too inefficient for the application update behavior envisaged. The updates observable by the kernel are at the granularity of a page. Page granularity is too coarse (e.g. the update may be a 4 byte pointer in a linked list node). To efficiently persist these kinds of transactions, the granularity should be as fine as a byte. The desired granularity could be obtained by adding system calls that support the specification of a transaction, but this would impose significant over-head, require POSIX approval and defeats the purpose of mapping the file in the application's address space.

Overview

Violet applications are written in C++ and specify transactions by leveraging an open API. A modified C++ compiler implementing the API compiles the application and instruments the code appropriately to execute the transactions. Finally, the application is linked with the Violet run-time library.

Violet's run-time system is implemented in a user-land library. Applications execute transactions and the library assembles the resulting updates and replicates them to a remote machine.

Transactional Memory

Applications are growing in size and complexity while the number of cores is increasing. As a result, the most common form of thread synchronization, locking, is growing increasingly more onerous. Lock hierarchies must be carefully designed and enforced to avoid deadlock. Selecting an appropriate granularity of locking is crucial to ensure the right balance is struck between parallelism and the cost in both time and space overhead. Priority inversion and lock retention across preemption can be serious artifacts when designing a system. Many data structures, such as balanced binary trees, are notoriously difficult to implement correctly while achieving reasonable performance with locks (e.g. the authors are unaware of any thread-safe red-black tree implementation that did not relax invariants to achieve satisfactory performance).

Herlihy introduced transactional memory [TM] as an alternative to locks to address the above concerns [7]. Transactional memory is a means of safely updating memory in the presence of concurrency that greatly simplifies code when compared with locks. In the last few years there has been strong of revival of interest in transactional memory (e.g. Intel's software TM compiler, GNU libtm). More recently, research database systems such as DBX [6] and HTM [23] have examined the implementation of databases with Intel's hardware transactional memory. Intel, HP, Oracle and others are proposing an update to the C++ language to incorporate TM directly into the language [21]. The proposed support exposes transactional memory as an integral language construct. Ephemeral DRAM data structures are the target use case. The abstraction proposed is of the form: *__transaction {}*. *__transaction* is a new C++ reserved word. The code between the braces would be executed transactionally with ACI (atomicity, consistency and isolation) semantics.

Violet leverages this new C++ reserved word as a means for applications to express relevant memory updates to the system; in effect the C++ proposal is Violet's API. Leveraging C++ increases the likelihood of Violet's adoption since C++ is not proprietary; it is an open standard. Moreover, applications are already being written with the proposed standard. Supporting the API makes the adoption of Violet for applications a *possibility* even as an after thought. Use of a modified compiler is a temporary measure that will be obviated by the implementation of the standard in clang (it is currently only supported in GNU's g++, which has licensing issues).

Violet Transactions

To execute transactions Violet leverages Intel's restricted transactional memory [RTM] feature [18]. RTM is included in Haswell processors' TSX facility. RTM transactions offer ACI semantics, but not durability; TSX is designed to work with ephemeral DRAM. Intel provides two instructions, *xbegin* and *xend*, that demarcate a transaction block. Applications begin a transaction by executing the *xbegin* instruction and commit by executing *xend*. The memory updates following *xbegin* are visible only to the executing thread until the execution of *xend*. Once a transaction commits then all of the updates to memory become visible simultaneously. If a thread writes to a memory location claimed by a peer thread's transaction then the processor aborts the transaction. The reliance on instructions introduced with Haswell TSX means that currently only Haswell TSX processors are supported.

The *xbegin* and *xend* instructions are a means of implementing a transaction, but the Violet library requires further instrumentation in the application. However, as we hope to support stock compilers in the future, reliance on modifications to clang++ must be kept to a minimum. clang++ was modified such that braces in the C++ *__transaction* construct correspond to *xbegin* and *xend* instructions; this is all that can reasonably be expected from a compiler. The remaining instrumentation of the code has to be done externally by a second tool that isolates the proprietary requirements. Violet uses a pre-assembler processor [PAP] to instrument application code. The tool operates on the assembly language emitted by the compiler. PAP scans the assembly code, identifies transactional blocks and then inserts the instrumentation. Transactional blocks are defined as everything in between *xbegin* and *xend* instructions.

The mechanics of a Violet transaction are depicted in Figure 2. The blue boxes are functions inside the Violet library. The arrows represent invocations inserted by the PAP; the application is not aware that they are there.

Opening Brace Execution

The opening brace results in the compiler emitting an *xbegin* instruction. Just after the *xbegin* instruction, a call to Violet's *start_tx* is inserted by the PAP. *start_tx* allocates a Violet transaction descriptor that is used to track the change-set of the transaction. A pointer to the transaction descriptor is placed on stack.

While every assignment between the braces is included in the transaction, not all are relevant to an application's persistent state. For instance, assignments to the stack are irrelevant. Only assignments to memory locations in the mapped file need to be replicated, and these are easily detected as they fall in the address range of the mapped region. The identification of the relevant change-set is done dynamically at run-time with instrumentation inserted at compile-time by the PAP. The PAP inserts an invocation to the *add_write_set* function before the assignments.

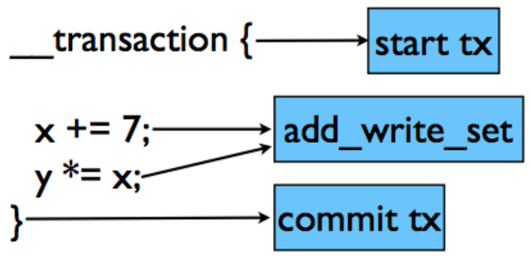

Figure 2: *A Violet transaction. The blue boxes are instrumentation added by Violet.*

Every memory assignment in the transaction block is passed through to *add_write_set* along with the pointer to the transaction descriptor. *add_write_set* identifies relevant addresses with a range check, and then records them in the transaction descriptor. Assignments in the x86 instruction set are of the form *movX*, where *X* is the type/size of the assignment. The type of the *mov* instruction is used by the PAP to determine the size of the assignment. Standard libc functions, such as memmove and strcpy, are not currently supported. Such functions are typically written in hand-tuned assembly and need to be made transactional. Moreover, the maximum size of a TSX change-set is finite and varies with many factors. Bulk data movement is probably best handled with different methods. If demand dictated then Violet versions of the functions could be included in Violet's run-time library.

Closing Brace Execution

A closing brace connotes a transaction commit and the compiler emits a *xend* instruction. The PAP will insert a call to Violet's *commit_tx* **outside** of the transaction block. A restriction of Intel RTM is that system calls (among other things) automatically abort transactions. *commit_tx* makes network system calls so it has to be called outside of the transaction block. This is a window of failure. The window of failure is on the order of a network round-trip-time. The RTM commits the transaction before it is replicated and made durable. Further, a successful transaction is also visible to other threads before it is made durable. Clearly, Violet is currently not suitable for applications that require ACID. We plan to address this in future work.

There are many ways to address the window of failure, such as a introducing some form of pre-logging, but we wanted to avoid imposing onerous burdens on the programmer or violating the C++ standard. To make the system as natural as possible to program, it was decided to identify the change-set automatically at run-time; the window of failure is the trade-off. Violet's window of failure is much smaller than the typical 30-second period between cleaning dirty pages in a buffer cache.

Cleanup of an aborted transaction is trivial. As no memory modifications made inside the transaction block are visible on abort, all of Violet's transaction metadata is cleaned up as a side effect of the failed transaction. While this artifact made implementation easier, it also made debugging difficult.

Replication

Following a successful RTM transaction, Violet replicates the results to a remote storage server. *commit_tx* is responsible for performing replication. The mutations to the file are recorded at the physical level, that is, the result of a transaction is recorded as the set of changed memory locations and their new contents.

An update to memory is encapsulated as a patch. A patch is a tuple consisting of a memory address, length and a string of bytes. Every memory location modified in a transaction is encapsulated in a patch. There is no type information.

The file is sharded over the replication group. The object of a replication group is to put more disks at the disposal of an application to increase disk bandwidth. Multiple disks do not increase reliability with multiple copies; they are there to increase disk bandwidth.

While there is a window of failure, Violet does guarantee that the remote copy of the file is always consistent. Consistency is enforced by the replication group. Memory updates are propagated to the replication group with a 2-phase commit protocol along with a monotonically increasing per transaction serial number. The serial numbers totally order the transactions. Their use is explained in §4.

The replication facility offers two modes of operation. Replication can either optimize network throughput or minimize the window of failure. If the application wishes to minimize the window of failure, the committing thread will wait for the result of the 2-phase commit before proceeding. It will also learn of any errors synchronously.

Far better use of the network is made if the committing thread simply queues its updates for transmission and carries on. The updates are transmitted when sufficient data have accumulated to fill an Ethernet frame. On a busy multi-core system, where transactions take on the order of 100 nanoseconds to complete, the wait time is usually sub-μs. We demonstrate in the results section that the window of failure is virtually the same in both modes of operation when commodity Ethernet is used.

4. Capacity Tier

In this section we describe the storage nodes (Sponges) used to persist the updates to the file on the host. We present a data structure, the Fibonacci Array, that is used to represent the file on disk. Finally, we show how the Sponges create distributed snapshots.

The capacity tier of Violet is comprised of a set of co-operating machines running a software agent called Sponge. It is a user level process. A Sponge is assumed to have at least one locally attached disk and some NVM (non-volatile memory) at its disposal. The Sponge is responsible for providing the capacity tier functions of disaster recovery and data management.

When a Sponge starts it determines what resources are available to it, such as the number of disks, and then it registers with the master server. The master server is discovered with mDNS. The master incorporates the Sponge into the cluster. Once a Sponge is registered with the master it is eligible for assignment to a file's replication group. Sponges can be members of any number of replication groups.

Mating the Host Update Behavior and Disks

A significant challenge for a disk based capacity tier is to mate the expected update behavior of the application on the host with the mechanical block-based world of the disk. Most applications today are conscious that they are backed by mechanical media and take great pains to interact well with them. DRAM data structures make no such concessions and focus on an asymptotic goal; DRAM data structures did not have disks in mind when they were developed. The updates to DRAM data structures can be `tiny' and there can be little locality. Furthermore, as applications are running at memory-bus speed, the rate of updates can be much higher.

In-memory applications only read the data on disk in failure scenarios, and then they stream the entire file. In the steady-state, the workload is write-only. Therefore, an index into the data is not required. An in-memory backing store just needs to build an image of the file from the incoming memory updates while keeping the cost of getting a memory update into the correct position in the file low.

Partial updates and poor locality are not new problems for storage systems. One technique of reducing the cost of an update is to amortize the cost of the I/O over multiple updates. An NVM staging area can be used to absorb updates and order them to detect opportunities for amortizing writes. Unfortunately, in the presence of very poor locality this strategy is not feasible. However, if the staging area is written out in its entirety and contiguously, then excellent amortization is realized; we call this staging with serial writing. In this way the random input stream is inflected into an ordered stream that leverages the streaming bandwidth of disks. Indeed, one can just keep doing this ad infinitum, but at some point the many disk resident logs need to be coalesced and reconciled rationally. A hierarchy must be superimposed on the disk resident pieces to regulate log resolution.

Two examples of disk log hierarchies are the LSM-Tree [12] and the COLA [2]. Both are excellent at dealing with random updates as they employ staging with serial writing. They also continually merge the disk resident pieces to incorporate updates yielding larger pieces. Both are optimized to keep an index into the data up to date. Because LSM-Trees and COLAs need to support lookup operations, the ratio of sizes between successive levels in their hierarchies is fixed to preserve properties required for search efficiency described below. In the absence of this requirement the ratio between levels can be varied permitting the merging of data through the hierarchy to its final resting place at lower cost.

Fibonacci Arrays

The Fibonacci Array [FA] is a data structure used to represent an in-memory file on disk. Each in-memory file has its own FA. The FA employs staging with serial write. The structure of an FA is depicted in Figure 3. It consists of an array of pointers to disk resident logs and a buffer, of size B, in NVM. The length of the levels in the array grows as a Fibonacci sequence; this is not by design but a natural artifact of the merging rule that is described below. The entries in an FA are patches containing memory updates.

The Fibonacci Array is derived from the COLA owing to its low amortized write cost. The array of levels in a COLA is managed like a binary number. The state of the array is a binary string where 1 indicates that a level is occupied and 0 denotes empty. For example, 1101 indicates that the first, second and fourth levels are occupied. A level, k, is either empty or contains 2^k entries. Flushing the NVM buffer to disk binary-adds 1 to the string. Merges occur when a binary carry is needed, e.g., flushing the NVM buffer to 1110 will result in a 4-way merge between levels 1-3 and NVM into the fourth level. The resulting string is 0001; so the 4th level is occupied and all beneath it are now empty. The following NVM flush will produce 1001, but no merge. In general, a merge, where k is the highest bit in the carry, requires $k2^k$ comparisons and a $(k + 1)$-way merge. This can lead to an overwhelming spike of I/O and CPU consumption as the height of the COLA grows. The COLA requires its merge rules to maintain invariants required to support efficient $\log_2$ search. The FA is free to use a different merge rule.

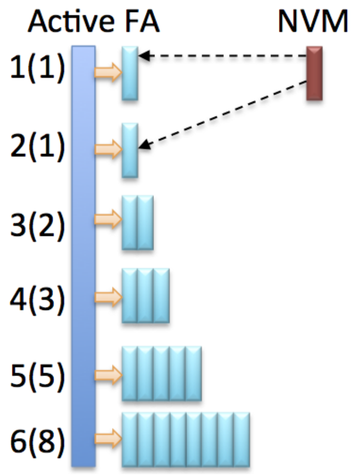

Figure 3: *Structure of a Fibonacci Array. Data start in NVM and then percolate down the array.*

The following rule is used to push data through a FA: when two neighboring levels in the array are occupied then they are merged into the next highest level. The source levels of the merge are left unoccupied. As the size of the merged level is the sum of the two source levels the array is naturally Fibonacci. Also, observe that the lowest source level skipped a level in the array and jumped two levels. Merging is done in the background. As only two levels are merged at any given moment the resource consumption of an FA is much smoother than a COLA. The 2-way merge permits the disk to seek less frequently and makes better use of the CPU.

Data propagate through a FA as follows. Updates arrive from the host in the form of patches that contain memory updates. As patches arrive they are placed in NVM in order of the memory location that they represent. Neighboring and overlapping patches are coalesced into a single patch where possible. As depicted in the diagram, once the NVM buffer is full it is written to disk. The system alternates between writing the NVM buffer to levels 1 and 2. When both are occupied they trigger the merging condition. As the logs are ordered, merging is trivially effected by streaming both source levels into the target level while maintaining the ordering. Neighboring patches are coalesced into a single entry as they are encountered. Patches can overlap producing a conflict. In this case the lower level's values are used as they are younger. Conflicting patches are resolved into a single patch. Applying the merge rule successively to all the levels of the array results in the data percolating down the array (driven by data flushing from NVM). At some point the data will arrive at the terminal level.

The terminal level in the array is the size of the file; a log cannot be larger than the file it represents. The terminal level will converge to a single patch (the length of the file). The height of the array, h, is approximately $\log_{1.61803}$(size of file), where 1.61803, the Golden Ratio, is the ratio between successive Fibonacci numbers. Thus, the per byte amortized cost of getting a byte to the terminal level is the number of times it was written over the size of the write: $0.75h/B$. The factor of 0.75 accounts for the fact that a datum skips a level on every second merge.

Fibonacci Array Snapshots

The FA includes a snapshot mechanism to support replication group snapshots. A FA snapshot is not exposed or created directly. It is the mechanism upon which consistent replication group snapshots are implemented.

A snapshot is created by copying the array of pointers into a new array that will become the active FA. Arrays and logs include a version number. When the new active FA is created its version number is incremented. As data percolate through the system and merging takes place, the mismatch between log version numbers and array version numbers is detected. A new log for the level is created with the active version number, and the pointer in the active array is updated.

FA snapshots are efficient because the active FA diverges from a snapshot at the granularity of the byte - exactly how the file actually is diverging. Explicit COW at the granularity of a page is supplanted by the temporal relationship between levels in the array. If a byte is updated and inserted into the active FA, it will enter the FA at the lowest level. Thus, the fact that it is the valid value in the active FA is implicit, and we do not need to know anything about what it over-wrote or where it is; nothing special has to be done as it is an intrinsic property of the data structure. This is efficient in both space and I/O.

Replication Group Snapshots

Snapshots can be taken of in-memory files stored in a Violet replication group. They offer a consistent view of memory as it appeared to the replication group when the snapshot was taken. An in-memory file stored in a replication group is distributed over more than one machine. A consistent snapshot therefore requires coordinating all of the Sponges in a replication group.

To create a snapshot of an in-memory file, the snapshot requestor queries the cluster master to discover the file's replication group. The requestor will then contact the Sponge with the highest IP address – this will be the leader for the snapshot. The leader contacts all of its peers and informs them of the snapshot request.

The replication group must come to a consensus on when the snapshot took place to ensure that the snapshot contains a consistent view of memory. Consistency is effected with the serial numbers of transactions. A serial number is agreed to such that every transaction that took place prior to it is in the snapshot and all those following are not. This serial number is the *snap-point*.

The host-side library aids in the determination of the snap-point. Whenever it sends a message to a Sponge it includes the latest *consistency-point*. The consistency-point is the highest serial number of a transaction such that there is an unbroken sequence of committed serial numbers back to zero. The leader solicits the highest consistency-point that has been observed by its peers. The highest consistency-point observed by any Sponge in the replication group is chosen as the snap-point. Note that during the course of this algorithm higher consistency-points may arrive on a Sponge, but they are ignored for the purposes of the current attempt of creating a snapshot. This ensures progress.

Once the snap-point has been published, Sponges proceed to create the FA snapshot. All patches in NVM that are in the snapshot are sent to disk. The FA snapshot is then created and the leader is informed. Once the leader has been informed of completion by all of its peers it informs the requestor.

5. Master Server

The master server is the glue that binds all of the pieces together. It is responsible for provisioning a file's replication group and file directory services. There is only one instance of a master in the system. While the master is a single point of failure there are many well known techniques to address high availability, e.g. a committee of machines running Paxos. As Violet is a research system the simplest implementation was adopted. Moreover, the master is not in the data-path and received little attention in this paper.

6. Results

We have implemented and evaluated Violet. The host-side library and tool chain were implemented on MacOS and Linux x86. The Sponge was implemented on

Linux x86. We used our prototype to evaluate the viability and effectiveness of Violet with commodity networks and disks.

The Violet data-path consists of two pieces: the host-side library and the Sponge. The master is not in the data-path and purely ancillary so it is not included.

The evaluation of Violet is decomposed into four parts. First, we measure the overhead of Violet instrumentation in application code to show that it is not onerous. Second, we explore how Violet interacts with a commodity Ethernet network when replicating. Third, we show that the Sponges efficiently create snapshots and restore data in the event of disaster recovery. Finally, we show that Violet can be used to implement DRAM data structures that are superior to standard storage data structures.

Instrumentation Overhead

We evaluated the overhead of Violet instrumentation on an Apple iMac configured with 16GB of DRAM. The processor is a 4-core Haswell i7-4771 with a clock speed of 3.5GHz. The i7-4771 supports TSX and hyper-threading.

Violet anticipates a new generation of in-memory DBs. As such there are no extant benchmarks or applications that are appropriate for the evaluation of Violet (nobody persists a C++ std::map today). As Violet was conceived for DRAM data structures we used two of the most common such data structures for Violet's evaluation: a self-balancing binary tree (a probabilistically balanced tree called a Treap) and a linked list.

To measure the cost of Violet's instrumentation a Treap was implemented with the proposed C++ TM standard. We measured its performance and compared it to the same code after the PAP was used to instrument it. Figure 4 presents the results.

The curve labeled Violet is a fully instrumented Treap. The RTM curve is the same code, but without Violet instrumentation. The times reported are the times taken to insert a single element in the Treap. The difference between the curves on the Δt axis is the temporal overhead introduced by Violet.

The initial knee in the curve results from going from a single thread to multiple threads. The time to insert an element remains roughly constant between 2 and 8 threads, as does the difference between the curves. In this range there is little pre-emption and CPU affinity is high. As the number of threads grows beyond 8 they start to compete for CPU time and pre-emption becomes a factor. Pre-emption causes RTM transactions to abort and the number of transaction retries begins to climb; this behavior is very different from locking where locks are held across pre-emption.

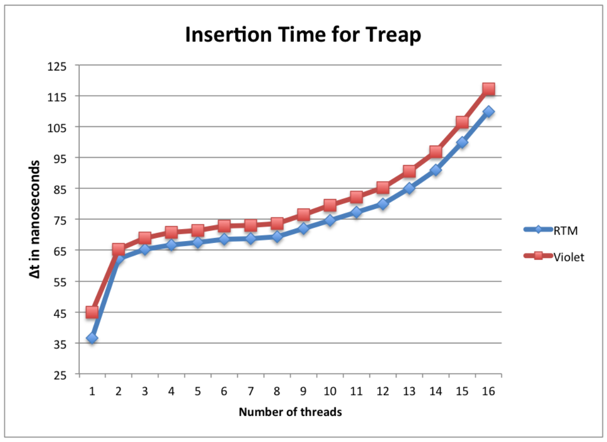

Figure 4: *Elapsed time for Treap insertions. Note that system call over-head is 200ns so far too slow for transactions on this time scale.*

The overhead of Violet's instrumentation appears to be constant, and quite low. Violet's instrumentation is very simple: two integer comparisons that comprise the address range check. The two integers that comprise the range are in the same cache-line and so popular that they are usually in the cache. Most of the expense of a transaction is incurred by loading the transaction's read-set from memory. All memory writes in a transaction are confined to the L1 cache until the transaction is committed.

Replication Performance

In this section we examine Violet's replication performance over commodity Ethernet. Applications that require replication but wish to run at CPU speeds will be highly sensitive to network performance.

A challenge when evaluating a system with a hardware dependency is gaining access to the hardware. The iMac is our sole TSX platform, but its only Ethernet option is 1 Gb. We felt that 1 Gb is too unrealistic for a modern server; a single Violet thread can saturate the iMac's NIC.

To perform experiments with 10 Gb Ethernet we used Amazon EC2. The EC2 `cluster instance' provides 10 Gb Ethernet, 2 Xeon processors (for a total of 8 cores), 1 disk and 60 GB of RAM. The `cluster instance' does not share processors; the processors are dedicated, but they do not support TSX.

The following experiments did not use TSX. Solely for this experiment the PAP removed the RTM instructions, but it produced fully instrumented code (that was not thread-safe). Threads were given exclusive access to private Treaps for correctness. The experimental setup consisted of a single host running the Violet Treap application replicating to a replication group that was configured with eight Sponges.

Figure 5 presents the aggregate throughput to the replication group as a function of the number of threads. We show two curves. The EC2 curve is the Treap application without RTM. This was run in EC2 and actually performed replication. The TSX curve is data produced by the iMac running fully instrumented RTM code, but not replicating (so just going as fast as it could). We include it to show that when Violet uses RTM it is still capable of driving an 8-node replication group.

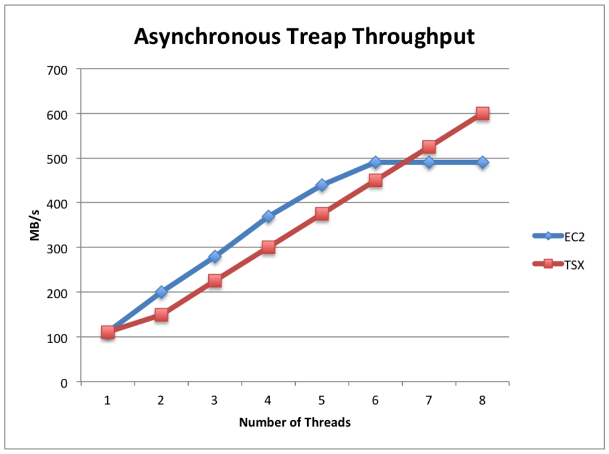

Figure 5: *Application insertion throughput***.**

EC2 throughput increases close to linearly until 490 MB/s is reached. This occurs when 6 or more threads are used. Benchmarking revealed that the EC2 10 Gb network is only capable of 490 MB/s so this is the saturation point of the network. If a superior network was available we believe that the throughput would have continued to grow because the Sponges had not yet reached their limit. The TSX curve intersects the EC2 curve where the latter hits the network's saturation point. This suggests that if the EC2 machines had been using RTM then the point of network saturation would have been postponed, as it would have been slightly slower.

The next experiment measures the performance of replicating synchronously. To demonstrate the sensitivity of replication to the choice of data structure an ordered doubly linked list was also used. Figure 6 presents the throughput for list updates being replicated both synchronously and asynchronously. The difference between the two is a factor of 2. Figure 6 also includes Treaps replicating synchronously. Asynchronously a single Treap thread produces ≈100 MB/s (Figure 5), but synchronously it slows down to 0.3 MB/s. This is a manifestation of the mismatch between the network's round-trip-time [RTT] and the time taken to commit a RTM transaction. The RTT in the EC2 network is 220 µs, and the time to execute a transaction is 45ns-120ns. A single transaction consists of only 40-80 bytes so the high latency is exacerbated by sending Ethernet frames that are practically empty. Lists do not produce as many updates as Treaps, threads spend more time looking for the insertion point in a list, so lists are not as sensitive as Treaps to synchronous replication.

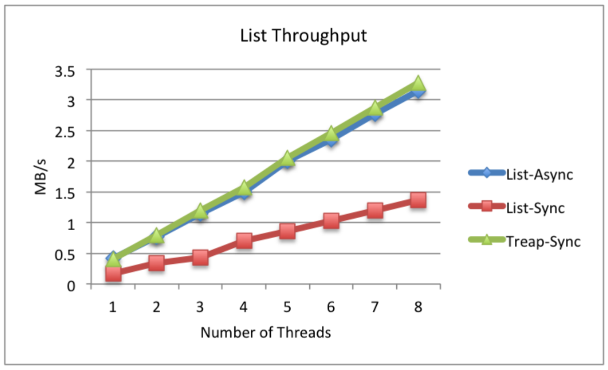

Figure 6: *List and Treap replication throughput*.

In the time it takes a Treap thread to synchronously replicate one transaction it could have committed ≈5,500 transactions. Moreover, 5,500 transactions would have filled many Ethernet frames in the same time frame thus triggering their transmission. We found that asynchronously replicated patches never waited longer than 1µs to be sent and on average waited 200ns. Thus asynchronously replicating updates offers roughly the same window of failure while dramatically increasing throughput. The throughput of synchronous replication is not viable over commodity Ethernet. For applications that can tolerate a small window of failure, we believe that Violet running with asynchronous replication is compelling. It is not uncommon for applications to tolerate dirty pages sitting in a kernel's buffer cache for 10's of seconds suggesting such applications exist.

Snapshot Creation Analysis

Snapshot performance as a function of the number of machines is depicted in Figure 7. A 25GB file was populated and then snapped. Taking a snapshot is quick operation requiring less than a second. The knee in the curve going from 1 to 2 machines results from consulting peer machines. A single machine can trivially find the snap point. When more than one machine is involved then the snap point must be found by inter-machine communication. Despite the knee in the curve, a replication group can create a snapshot in less than a second.

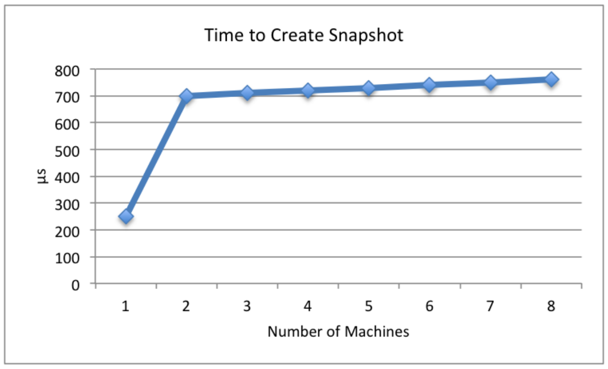

Figure 7: *Time required to take snapshot as observed by requestor***.**

Restore Analysis

To be useful for disaster recovery the system has to be able to restore a file in a sufficiently `short' period of time. In an environment such as EC2 the fastest one can restore a file is constrained by the network connection between the machines. The top throughput of EC2's network is 490 MB/s. The question then becomes how many Sponges does it take to saturate the network (this is the time to restore).

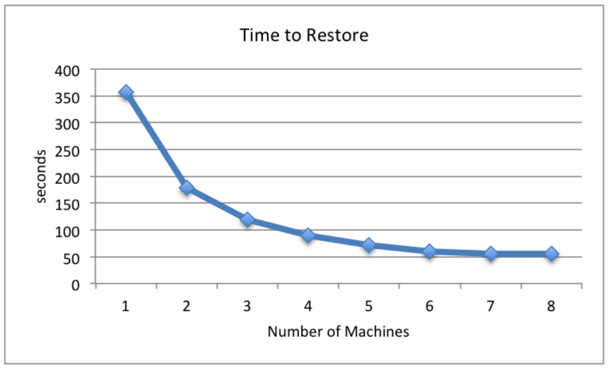

Figure 8: *Time to restore as a function of Sponges/disks***.**

Restore performance is presented as a function of the number of machines in Figure 8. The size of the file is 25 GB. As can be seen from the graph, the Sponges saturate the network at 6 machines. When the network is saturated it takes 53 seconds to restore a 25GB file. The individual Sponges are always disk bound and manage ~80 MB/s each until the network is saturated at 6 machines. This is close to the peak read rate observed in the cluster instance (90 MB/s). The contiguous disk layout of the FA saves the disk from seeking frequently.

Asymptotic Driven Data Structures

In-memory computing has vastly increased the speed of analytics. However, much of the advancement has come from the improvement of the inherent characteristics of the media, by using DRAM instead of disks. Merely swapping the media type only partially realizes the potential speed-up if the same data structures continue to be used. For example, SAP HANA is an in-memory DB, but for the most part it is columnar. This is not very different from deploying a columnar DB on a RAM disk. Merely exchanging media ignores many of the other advantages of DRAM; e.g., it offers random byte-grained access. The adoption of DRAM suggests that different data structures could be used thus also realizing an *asymptotic* speed improvement.

The greatest potential for performance improvement of in-memory DBs lies in adopting data structures that are usually overlooked because they are difficult to persist: DRAM data structures. DRAM is volatile so in-memory databases must still persist their state to block devices. An obstacle to the adoption of more complicated data structures is the difficulty in persisting them correctly and efficiently. Violet was developed to bridge this gap.

Employing domain specific databases has been suggested in the past [8, 9] and shown to be superior. To motivate this argument we present two queries that are important to our customers that they typically run on columnar DBs. For our experiment both queries were run on MonetDB [3] provisioned with a RAM disk. The queries were also run on a domain specific data structures created with Violet. The point is not to demonstrate that the Violet system is faster per se (it is not a fair comparison), but to demonstrate how poor linear scans are for many problems important to our customers. If databases are produced that can cut hours off of computation they will be adopted [8, 9].

The first query that we present is the identification of clusters of points in a data set. Cluster identification is an important knowledge discovery technique, e.g. market researchers interpret the clusters as market segments. An important algorithm for this application is DBScan [4]. DBScan runs in $O(N^2)$ time with a columnar DB, but it runs in $O(N)$ time when the DB supports a nearest neighbor query. The Voronoi diagram supports the nearest neighbor query so it was implemented with Violet. The results are shown in Figure 9; the predicted difference in growth of run-times is observed (note the log scale).

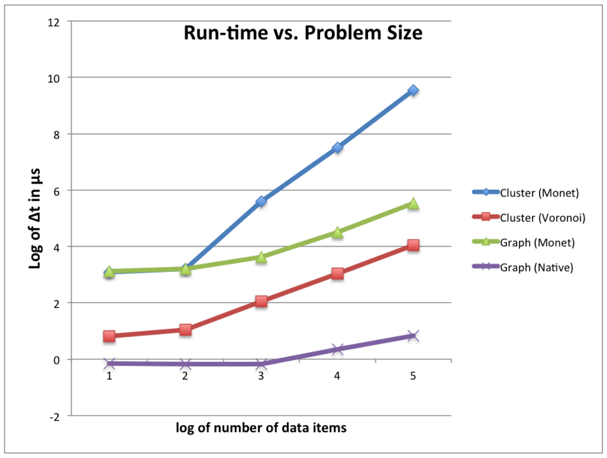

Figure 9: *log/log scale asymptotic behaviour of clustering and path queries***.**

The second query that we examined is a path query in a graph. Graphs arise naturally in a lot of applications and they have been attracting a lot of attention in analytics; e.g., SAP is embracing graph computing by grafting a graph query engine on to HANA [17]. A columnar DB, however, is not the most natural way to represent a graph in memory. We implemented a graph DB that represents the graph as memory nodes and pointers. Queries on a graph take *O(N)* time in a columnar DB, but only grow by the diameter of the graph when represented naturally. Figure 9 presents the observed empirical asymptotic superiority of the natural representation.

In both implementations of the domain optimal data structures there is a gross asymptotic advantage hence a strong motivation to adopt them. Violet made it easy to persist the data structures as no thought had to be given to the persistence layer; a non-trivial problem in the case of a Voronoi diagram. Violet just took care of it. Implementation effort was directed at a correct implementation of the data structure.

7 Related Work

SNIA non-volatile memory (NVM) working group has proposed a standard for accessing new types of non-volatile memories [16]. Currently, they have proposed both a kernel level volume, and a kernel level file interface for accessing NVM at a fine-grained level. They are also interested in proposing a user space level interface to allow applications to access NVM at a fine-grained level using the load and store data access model. They also want to introduce the notion of transactions for this user level API. The work that has been proposed in this paper can be leveraged by this SNIA working group.

Different data management middleware offerings like Redis [15], SAP HANA [11], Microsoft_Grace [19], Facebook_TAO [20] are proposing strategies for data management at the data structure level (e.g. graphs, KV stores). Redis allows for the persistent management of key-value storage data structures. Redis provides off-node data protection by copying the data at a file level. SAP Hana allows for in-memory manipulation of graph data structures and it maps these data structures on to a columnar database that, in turn, moves data off-node at page level granularity [17]. Microsoft Grace system stores the graph data structures such as vertices and edges in respective files on disk, and subsequently it reads these structures in parallel when loading in the graph. Additionally, Grace also maintains a log of committed updates on disk. Facebook TAO is a geographically distributed eventually consistent graph store. TAO shards the dataset into shards and stores these shards across multiple database servers. TAO also maintains an elaborate leader-follower caching infrastructure in front of its persistence layer, and it uses an eventual consistency model to keep the data consistent amongst the caches.

Recoverable Virtual Memory [22] addresses a similar space as Violet, but takes a different approach. RVM employs explicit logging and requires the programmer to identify and backup copies of data. Concurrency is consciously left to the programmer to address separately. The file is updated with a naïve staging-and-write strategy. Consistency in the file is maintained with the log and write-ordering the file's dirty pages.

The work being proposed in this paper is independent of the type of data structure being supported in memory. That is, we detect updates to any type of data structure at a fine granularity and then subsequently ship these fine-grained updates off node and stream them on to a block based back-end storage device. Thus, our work can be leveraged by the above mentioned middleware systems.

8 Conclusions

In this paper we describe a storage system that spans across the host and a disk-based backend storage system. This architecture helps to map fine-grained host side data management with a block level data management system at the backend. The techniques presented in this paper become important as real-time analytics applications begin to employ data structures that have been designed with main-memory computing model in mind and that want to backup these data structures in an efficient manner on to a cheaper off-box disk based storage system. We propose a host side user space client library that leverages the CPU's transactional memory instructions to efficiently detect fine-grained updates. We also present a new data structure called a Fibonacci Array at the backend disk subsystem that helps to stream update operations on to a disk in a more efficient manner by optimizing the data structure for primarily write operations. The Violet system divorces the implementation of a data structure from its persistence. Finally, we implemented our ideas in a prototype and demonstrated the benefits of managing in-memory data structures natively, rather than mapping them to intermediate data stores that are not designed to deal with in-memory data structures natively.

Bibliography

[1] F. Aurenhammer. "Voronoi diagrams: A survey of a fundamental geometric data structure." In ACM Computing Surveys, 23(3):345–405, 1991

[2] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. "Cache-oblivious streaming B-trees." In SPAA, 2007

[3] Peter Boncz, Stefan Manegold and Martin Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", In VLDB, 1999

[4] M. Ester, Hans-Peter et al (1996). "A density-based algorithm for discovering clusters in large spatial databases with noise", Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996).

[5] Rich Freitas and Larry Chiu, "Storage Class Memory: Technologies, Systems, and Applications." In USENIX FAST 2012 Tutorials

[6] Zhaoguo Wang, Hao Qian et al, "Using Restricted Transactional Memory to Build a Scalable In-Memory Database," Eurosys 2014

J.Guerra, L.Marmol et al, "Software Persistent Memory," In USENIX ATC 2012

[7] M. Herlihy and J. E. B. Moss. "Transactional memory: Architectural support for lock-free data structures." In Annual International Symposium on Computer Architecture, 1993.

[8] M. Stonebraker and U. Cetintemel. "One Size Fits All: An Idea whose Time has Come and Gone." In ICDE 2005.

[9] M. Stonebraker, Nabil Hachem et al., "The End of an Architectural Era (It's Time for a Complete Rewrite)." In VLDB 2007

[10] M. Stonebraker, "New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps", in Communications of the ACM, June, 2011.

[11] F. Farber, S. Cha, J. Primsch, C. Bornhovd, S. Sigg W. Lehner. "SAP HANA Database: Data Management for Modern Business Applications", in SIGMOD Record, January, 2012.

[12] O'Neil, P. E., Cheng et al. "The log-structured merge-tree (LSM-Tree)." in Acta Informatica 33, 1996

[13] "Oracle TimesTen In-Memory Database on Oracle Exalogic Elastic Could", Oracle White Paper, July, 2011.

[14] "VoltDB: High Performance, Scalable RDBMS for Big Data and Real-Time Analytics", White Paper, 2012.

[15] Jeremy Zawodny, "Redis: Lightweight key/value Store That Goes the Extra Mile", Linux Magazine, August 31, 2009

[16] SNIA NVM Programming Model, version 1.0.0 Revision 5, Working Draft, June 12, 2013

[17] M. Rudolf, M. Paradies, C. Bornhovd, and W. Lehner. "The Graph Story of the SAP HANA Database", in BTW, 2013.

[18] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Document Number: 248966-028 July 2013

[19] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou and M. Haridasan, "Managing Large Graphs on Multi-Cores with Graph Awareness", USENIX ATC, 2012.

[20] N. Bronson et al, "TAO: Facebook's Distributed Data Store for the Social Graph", USENIX ATC, 2013.

[21] Transactional Memory Support for C++, www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3859.pdf

[22] M. Satyanarayanan et al, "Lightweight Recoverable Virtual Memory," SOSP 1993

[23] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In Proc. ICDE, 2014.