Protea: An Abstraction for Building Flexible Storage Systems

Jason Kim and Harvey Xia

December 22, 2015

Abstract

Protea is a library and interface for building general storage systems. It attempts to capture the heterogeneity of existing storage systems as well as hypothetical systems and their underlying hardware in a unified and simple tree representation, the **ProtoTree** where different *operator nodes* abstracting over common storage functional idioms are composed together on top of a set of *device nodes* representing storage media. Protea aims to enable simpler and faster design and implementation of storage systems.

Keywords. storage systems

1 Introduction

Today system designers are faced with a wide array of different storage hardware. Table 1 shows a subset of the variety of storage devices that exist today, with their associated performance metrics and cost. NAND flash has seen wide adoption due to its advantages as a persistent storage medium over regular magnetic disk storage. NVRAM exists as a non-volatile alternative to DRAM. Shingled Mangetic Recording (SMR) has improved storage density compared to regular HDD's. The problem of designing a storage system is no longer a simple process of employing the default de facto storage hierarchy. System designers can explore the complex tradeoffs between throughput, latency, read vs. write performance, and cost, among other relevant properties. Indeed, many papers of the past decade have proposed novel storage systems that exploit the unique characteristics of certain storage hardware to produce favorable properties for the entire system.

Protea aims to capture the diversity of existing heterogeneous storage system designs and enable rapid construction of hypothetical heterogeneous storage systems in a novel, unifying abstraction. This should allow for simpler implementations of existing systems and facilitate the realization of novel designs. Our approach is motivated by the observation that existing storage systems are compositions of a common set of *functional idioms*. Such idioms include caching for faster reads and writes, logging for write performance, striping data for higher throughput due to parallelism, and mirroring for reliability and durability.

Devices	Throughput	Latency	Cost / GB
Registers	-	1 cycle	-
Caches	-	2-10 ns	-
DRAM	10s of GB/s	100-200ns	$10
NVDIMM	10s of GB/s	100-200ns	$10
NVMM	10s of GB/s	800ns	$5
NVMe	2GB/s	10-100s	$1.4
SATA SSD	500MB/s	400s	$.4
Disk	100MB/s	10ms	$.05

Table 1: modern memory hierarchy

Thus Protea consists of a library of *operator nodes* and an *interface*. Operators are abstractions over the common functional idioms mentioned above. Protea implements them so they can be reused and defines an interface by which they can be composed together in a well-defined manner.

Section 2 describes the abstraction of the ProtoTree. Section 3 describes a candidate design of the ProtoTree.

2 The ProtoTree Abstraction

We use the tree data structure to model a storage system. The leaves of the tree are *device nodes* representing hardware storage devices. Intermediate nodes are *operator nodes* that represent common functional idioms in storage systems, such as logging, indexing, read caching, write caching, striping, and mirroring. Operator nodes adhere to a common read-write interface, allowing them to be composed together in well defined ways.

The ProtoTree abstraction is motivated by a *modular* approach to building storage systems. The hypothesis is that existing storage systems can be thought of as a composition of many modular components. Furthermore, there is an implicit hierarchy to this composition. For instance, a write cache manages the behavior of its cache and backing storage. Therefore a tree seems like the natural way to capture both the modularity and hierarchy of storage systems, with nodes representing storage abstractions / devices and edges representing the various

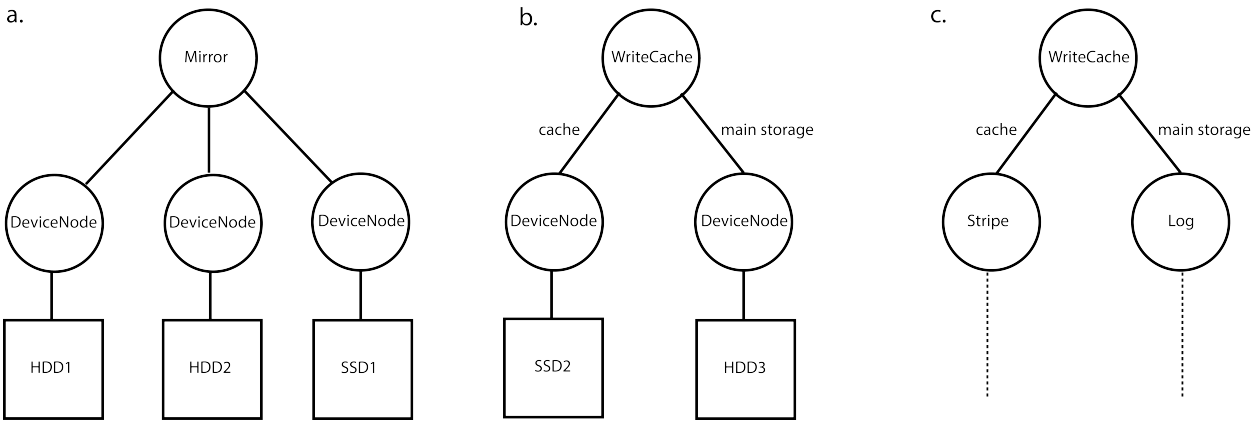

Figure 1: Examples of simple ProtoTrees.

possible relationships between abstractions and devices.

An additional benefit of the ProtoTree design is that it encourages application programmers to think about their system in a modular fashion. This ensures that all relationships between various storage devices and abstractions are made explicit, preventing unanticipated behavior resulting from unexpected interactions between storage abstractions from occurring.

Protea also aims to enable imposing ACID semantics on the system. For instance, programmers should be able to specify that writes to a given operator node are atomic and durable. The ProtoTree should also be able to recover from various failure modes.

For instance, Figure 1.a shows an intermediate mirroring node sitting on top of three device nodes. A write operation on the mirror node would result in a write operation being issued to each of the device nodes, which in turn write to the two hard drives and the one solid state drive. Figure 1.b shows a different scheme in which a solid state drive acts as a cache for a write cache, and a hard drive acts as the main storage. Figure 1.c shows a more complicated scheme in which the cache child of a write cache is another intermediate node. Under this scheme, data would be cached by striping it across the children of the stripe node. Although complications due to edge case interactions between intermediate node are a concern, the behavior of composed intermediate nodes should be largely predictable and intuitive.

In addition to implementation choices, this abstraction raises many design choices. What should the interface between nodes be? How can different intermediate nodes interact? How is state stored for intermediate nodes? What are the semantics for ACID properties on the various intermediate nodes? These questions are explored in the following section. We discuss our approach to these questions in the following section.

3 A Candidate Design

3.1 DeviceNodes and BlockDevice

Recall that device nodes are leaf nodes sitting immediately above the hardware medium, i.e. they contain logic that directly interacts with device drivers. Currently the system relies on the Linux file interface exposed through the /dev directory to interact with devices. This method is less than ideal because performing I/O using a device's file handle by default employs operating system buffering. This can be avoided by passing the `O_DIRECT` flag to the native C `open()` system call, but Java does not support this feature and we have yet to implement a Java based version.

Device files expose a byte-addressable space. The DeviceNode class performs I/O on device files through Java's `read(byte b[], int off, int len)` and `write(byte b[], int off, int len)` file object methods. An additional class, BlockDevice is layered on top of DeviceNode in order to support block addressable devices. The BlockDevice class is initialized with a grain such as 4KB as its block size, and IO is performed by block rather than by byte. Values are padded as needed in order to align them to the block boundaries.

3.2 Operator Nodes

In our implementation of the ProtoTree, each operator node adheres to the same key-value map interface shown in Listing 1, exposing `get(long key)`, `put(long key, byte[] value)`, and `delete(long key)` methods. Under this design, all data is associated with a 64 bit key and are written and read using those keys. This interface simplifies reasoning about the system, as all references to the same data are associated with the same key, and all operator nodes interact with each other

through calling put(), get(), and delete(). Certain operator nodes interpret the key as an address, but this will be discussed later. Also associated with each operator node is an ID assigned by the application programmer. This ID is used to uniquely identify the given node, as well as generate log files for that node. Operator nodes take other operator nodes as children.

ReplicaNode: The ReplicaNode implements RAID1's mirroring functionality. It has a variable number of children, each of which represent a single replica on which to mirror data. On receiving a put() request, the ReplicaNode simply performs a put() on each of its children. On receiving a get() request, the ReplicaNode calls get() on the single primary child node. In the future, additional logic can be added to improve availability and performance for concurrent access, i.e. if multiple clients issue a get() to the ReplicaNode.

StripeNode: The StripeNode implements RAID0's striping functionality. It has a variable number of children, each of which represent a single node on which to stripe data. The key space is striped evenly across children by taking the key modulo the number of children. Thus, it is up to the application programmer to assign their keys in such a way as to ensure even distribution across children.

ReadCacheNode: The ReadCacheNode implements a read cache, using its left child which must be a LogNode as the cache for its right child. The ReadCacheNode contains local state consisting of a hash set used to determine existence of a key in the cache. On a get() request, a key is retrieved from the cache if it exists there, otherwise it is retrieved from the backing storage. Currently the ReadCacheNode implements the most naive migration policy, it migrates when the cache is full. This node can be improved by allowing the programmer to inject custom migration policy logic. This can be done by creating an interface containing abstract methods called by ReadCacheNode for determining when and how to migrate. The application programmer implements the interface and declares that it is to be passed into the initialization of ReadCacheNode.

WriteCacheNode: Similar to the ReadCacheNode, the WriteCacheNode implements a write cache using its left child which must be a LogNode as the cache for the right child. It also maintains a local hash set for determining existence of keys in the cache. The migration policy is also to migrate when the cache is full.

SimpleMapper: The SimpleMapper is simply an in-memory key-value hash map. It is useful if the application programmer requires an in-memory hash map for storing custom local state.

```
class Node {
    String id;
    byte[] get(long key);
    void put(long key, byte[] value);
    void delete(long key);
}
```

Listing 1: Operator node interface

3.3 Lowest Tier Operator Nodes

Only a subset of the operator nodes can have DeviceNodes or BlockDevices as children. As the system currently stands, the PassThroughMapper and LogNode interface with devices.

PassThroughMapper: The PassThroughMapper is simply a pass-through map over a BlockDevice. Here the 64 bit keys are interpreted as block addresses. An error is thrown if the key exceeds the boundary of the BlockDevice.

LogNode: The LogNode is a log-structured key-value store over a block address space. The LogNode has two children, an implicit RAM node that stores an index containing keys and their associated address in the BlockDevice, and the BlockDevice on which IO is performed. When receiving a new put() request, the LogNode appends the value to the tail end of the BlockDevice and adds an entry to the index with the key and the address of the associated value. To serve get() requests, the LogNode finds the address of the requested key and performs a read at that address and returns the value. The LogNode contains local state consisting of a pointer to the tail of the log where new data is appended and a list of deleted entries to be garbage collected.

As the system currently stands, all IO over BlockDevices must be through either a PassThroughMapper or LogNode. As discussed in the classical LFS paper[5], structuring the disk as a log leads to significant speedups for writes. Garbage collection on the LogNode is currently not implemented. In the future, we plan on implementing a lowest tier operator node that allocates storage space using traditional non log-structured techniques used by existing file systems.

3.4 Atomicity and Durability

A user may require operator nodes to persist their metadata or perform atomic operations (i.e., transactions). To facilitate these requirements, our implementation allows users to flag operators as atomic. When an atomic operator performs a transaction, it first appends a `beginTX` record to a write-ahead log and then appends a record to the log for each operation performed within the transaction. When the transaction is complete, the operator finally appends a `commitTX` record to the log. In this

way, if the system crashes before a transaction is committed, then upon recovery it will know to undo any uncommitted operations issued on the atomic operator. Because every record in the log is tagged with a transaction ID, concurrent transactions will still behave atomically.

Write-ahead logging also allows us to recover metadata after a system crash. Because atomic operators play back all committed operations on the log, the operator will not only recover its pre-crash hard state but also its soft state. Currently, the system does not make use of checkpointing, so upon system crash, each log must be replayed all the way through to ensure that state is fully recovered. However, adding checkpointing as a feature would be straightforward. A nave implementation would consist of periodically snapshotting the system state and asynchronously writing to disk. Only two checkpoints would be required to ensure that the system can always recover in a timely fashion. The older of the two checkpoints would be overwritten so that in the event of a crash during recovery, the system can still recover from the newer checkpoint and any subsequent transactions on the log.

3.5 Declaring ProtoTrees

Application programmers construct ProtoTrees in a declarative manner, specifying the structure using JSON (JavaScript Object Notation). The root level object represents the root node of the ProtoTree. Each level of the JSON object must contain a `type` field specifying the type of the node, an `id` specifying the unique ID for the node, and `atomic` flag specifying whether the subtree rooted at this node should be atomic. Depending on the type, the node may need to include a `children` array or `left` and `right` fields. Certain nodes, such as the ReplicaNode and StripeNode take a variable number of children. Other nodes, such as the WriteCacheNode and ReadCacheNode take a left child as the cache and the right child as the backing storage.

All subtrees must terminate with a LogNode or PassThroughMapper, each of which requires not only a type and ID, but the Linux file device handle beginning with "/dev" as the `deviceName`, the `blockSize` of the device, and the `deviceSize` or capacity. All sizes are specified in bytes.

JSON notation is well suited as a declarative format for the ProtoTree because its ability to nest arbitrarily can represent different levels of the tree. Furthermore, it is a widely adopted format and one that nearly all programmers are familiar with. Declaring ProtoTrees using JSON is compact, simple, clear, and portable, which we anticipate will facilitate collaboration and sharing of ProtoTree designs.

As the system evolves, if additional metadata is required, additional fields can be required of the JSON file. At run time, JSON files are validated for the correct fields and types. Valid JSON files are then processed recursively and a ProtoTree object is generated.

```
{
  "type": "replicaNode",
  "id": "replicaNode1",
  "atomic": false,
  "children": [{
    "type": "logNode",
    "deviceName": "/dev/xvdb",
    "id": "hdd1",
    "deviceSize": 160000000000,
    "blockSize": 4000
  }, {
    "type": "logNode",
    "deviceName": "/dev/abc",
    "id": "ssd1",
    "deviceSize": 4000000000,
    "blockSize": 4000
  }, {
    "type": "logNode",
    "deviceName": "/dev/abd",
    "id": "ssd2",
    "deviceSize": 4000000000,
    "blockSize": 4000
  }]
}
```

Listing 2: An example ProtoTree JSON for a ReplicaNode

3.6 Application Usage

The current implementation is in Java. As Listing 4 illusrtates, application programmers download the Protea Java package and import it where needed, then calling the `generateTree()` method while passing in a JSON declaration, which then returns the node at which the declared ProtoTree is rooted.

```
import com.github.projectdelos.Protea

Node tree = Protea.generateTree(json);
tree.put(key1, value1.getBytes());
```

Listing 3: Example usage of the Java implementation of Protea

4 Example ProtoTrees

```
{
  "type": "writeCacheNode",
  "id": "griffin",
  "atomic": false,
  "left": {
    "type": "logNode",
    "deviceName": "/dev/xvdb",
    "id": "hdd1",
    "deviceSize": 64000000000,
```

```
    "blockSize": 4000
  },
  "right": {
    "type": "logNode",
    "deviceName": "/dev/abc",
    "id": "ssd1",
    "deviceSize": 32000000000,
    "blockSize": 4000
  }
}
```

Listing 4: Example ProtoTree for the Griffin system

In terms of applications that utilize Protea, Griffin[6] is a good candidate. Griffin is a hybrid storage system that uses an HDD as a log-structured write cache for SSD. Its goal is to minimize writes to the SSD while maintain read performance, thus conserving SSD erase cycles and prolonging its lifetime. Writes are logged sequentially to the HDD and periodically migrated to the SSD. Reads are usually served from the SSD and occasionally from the slower HDD. Listing 2 shows a Griffin-like ProtoTree. The left child of the writeCacheNode is a hardrive cache and the backing storage is a solid-state drive. It can be used as it stands as a simple key-value store. Or the application programmer can write her own custom upstream interface.

```
{
  "type": "simpleMapper",
  "id": "simpleMapper1",
  "deviceSize": 4000000
}
```

Listing 5: JSON for simpleMapper

```
{
  "type": "logNode",
  "deviceName": "/dev/abc",
  "id": "ssd2",
  "deviceSize": 16000000000,
  "blockSize": 4000
}
```

Listing 6: JSON for logNode

Corfu[2] presents a more concrete example of a distributed storage system, that of a shared log implemented over a distributed set of SSD's, designed to work over dumb network-attached flash devices. Each client maintains a consistent map of positions in the log to flash pages on different flash units. To read, the client looks up the corresponding flash device of a particular log position and then directly issues a read to that device. Similarly to append, the **sequence node** gives the client the next available position in the shared log, and the client writes data directly to the corresponding set of flash pages. Each constituent flash unit is a write-once address space that returns errors on reads to unwritten or trimmed slots, and writes to written or trimmed slots. Trimming an address prevents it from being used again. This type of flash unit can be implemented using Protea.

A custom API can be constructed on top of two ProtoTrees, a simpleMapper and a logNode as shown in Listing 5 and Listing 6. The simpleMapper maintains metadata, mapping each address to one of three states: "written", "unwritten", and "trimmed". The API exposes `write(address, value)`, `read(address)`, and `trim(address)` methods. To serve a write, the simpleMapper is consulted to determine the state of the address. Either an error is returned or the write is performed on the logNode and the simpleMapper updated. Read and trim operations are performed in a similar manner, throwing errors when necessary.

5 Future Work

Protea offers a useful framework not only for thinking about and implementing storage systems but also for automated tree discovery. Due to the modular design of ProtoTrees and the simplicity of implementing new designs, one could use an evolutionary algorithm to generate trees that demonstrate desirable characteristics. For example, if a user wants a ProtoTree design that handles read-heavy workloads and utilizes an HDD and SSD, they could iterate through generations of random tree candidates and select for those that demonstrate the best read throughput. This inverts the normal process of storage system design, in which arduous human-driven design comes first and then evaluation. Instead, such an automated discovery framework opens up the possibility of machine-driven design and evaluation, where the human only needs to specify their goals.

Another direction for this work is to implement Protea on an FPGA so that, by ditching an operating system entirely, the system can benefit from significant speedup, pushing the speed bottleneck to network and disk I/O.

6 Related Work

Griffin and Corfu are two fitting candidates for implementation via Protea. They are discussed in section 4.

The Arrakis[4] operating system gives applications direct access to virtualized I/O devices, eliminating the overhead associated with kernel mediation. The notion of eliminating the traditional role of the kernel in managing I/O operations is relevant here because Protea acts primarily as a data store, its functionality almost limited exclusively to abstractions on top of I/O operations. Furthermore, ideally Protea would be implemented on FPGA's that provide specialized hardware support. Our project, however, focuses on proving the Protea concept on the software side. Later iterations of the project might

involve stripping away kernel overhead in a fashion similar to Arrakis.

MosaStore[1] presents the idea of aggregating node-local resources (i.e. storage space, IO channels, memory, etc.) across a distributed network and utilizing them for a dedicated storage system optimized for the particular application's workload. In MosaStore, *Donor nodes* donate storage space to the system, as managed by a centralized *metadata manager*. Each client on the network installs a *system access iterface* library, giving them an interface with which to access the distributed storage space. The aspect of *specialization* is similar to the *flexiblility* design goal of Protea, where the underlying storage system is tailored for the custom API of the application. It might be interesting in future iterations of Protea to explore the possibility of extending it over a distributed network of storage servers or network-enabled hardware.

Work has been done on designing heterogeneous storage systems to augment OS functionality. Kim[3] proposes a hybrid flash architecture that uses PRAM (phase-change ram) for storing file system metadata and regular NAND flash for user data storage. PRAM possesses fast byte access and does not require erasure before overwriting, making it more suitable for file system metadata which is frequently modified in small sizes. This architecture also extends the lifetime of the NAND flash devices.

References

[1] AL-KISWANY, S., GHARAIBEH, A., AND RIPEANU, M. The case for a versatile storage system. *SIGOPS Oper. Syst. Rev. 44*, 1 (Mar. 2010), 10–14.

[2] BALAKRISHNAN, M., MALKHI, D., DAVIS, J., PRABHAKARAN, V., WEI, M., AND WOBBER, T. Corfu: A distributed shared log. *ACM Transactions on Computer Systems* (2013).

[3] KIM, J. K., LEE, H. G., CHOI, S., AND BAHNG, K. I. A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM International Conference on Embedded Software* (New York, NY, USA, 2008), EMSOFT '08, ACM, pp. 31–40.

[4] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 1–16.

[5] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[6] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 8–8.