Protea: An Abstraction for Building Flexible Storage Systems

Harvey Xia
Advisor: Mahesh Balakrishnan
December 22, 2015

Abstract

System designers today have a wide selection of storage devices (flash memory, NVRAM, disk, etc.) at their disposal, each with distinct characteristics and tradeoffs. Substantial work has been done in investigating novel storage system designs that optimize for different properties, such as Griffin[6] which uses a hard drive as a write-cache to extend the lifetime of the backing SSD. A result of the proliferation of hardware devices is an increase in flexibility as well as complexity when it comes to designing storage systems. To facilitate the process of designing storage systems we propose Protea, an interface and library of storage idioms for building storage systems.

Protea is a library and interface for building general storage systems on a single machine connected to a heterogeneous set of hardware storage devices. It attempts to capture the heterogeneity of existing and hypothetical storage systems and their underlying hardware in a unified tree representation, the ProtoTree. Under the ProtoTree abstraction, storage systems are represented by operator nodes and device nodes. Operator nodes abstract over common storage functional idioms and 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 Magnetic 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 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 allows for simpler implementations of existing systems and facilitates 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.

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. Section 4 describes some example systems implemented in Protea. Section 5 describes an evaluation of Protea. Section 6 describes related works. Section 7 presents concluding remarks and future directions for Protea.

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

Table 1: modern memory hierarchy
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.

2.1 Modularity

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 both storage abstractions and devices and edges representing the various possible relationships between abstractions and devices.

The ProtoTree design 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.

2.2 ACID Semantics

Protea also aims to enable imposing ACID semantics on the system. A natural notion of a transaction emerges from the nature of each type of node. We discuss an initial implementation of atomicity and durability in section 3.5. The ACID properties should be exposed to the application programmer for greater flexibility. 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? Should this interface be uniform between different nodes? 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 Key-value Interface

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

Listing 1: Operator node interface

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. A key-value style put-get interface is a minimal way to express the core primitive operation of Protea, that of storing and fetching data. Under this design, all data is associated with a 64 bit key and are written and read using those keys. A nice property of this interface is that a key acts as a common handle on a single piece of data across the entire storage stack. This facilitates keeping track of data.

This interface also 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() and get(). Certain operator nodes interpret the key as an address, but this will be discussed later.

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 unique filenames for the log files of that node. Operator nodes take other operator nodes or device nodes as children.
### 3.2 DeviceNode 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 through the Java Native Interface. Given that the ultimate target platform for Protea are FPGA’s, implementing direct IO on UNIX based operating systems is not an urgent priority.

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.

Both DeviceNode and BlockDevice are provided as the two alternative low-level abstractions for accessing storage. This allows the programmer flexibility depending on whether a certain aspect of the storage system requires a byte-addressable space or a block-addressable space.

### 3.3 Operator Nodes

**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 in a custom class, instantiates that class, 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 deter-
mining existence of keys in the cache. The migration policy is also to migrate when the cache is full.

Similar to the ReadCacheNode, the WriteCacheNode implements naive placeholder migration logic. Future work would be to allow application programmers to inject custom cache policy logic suited to their application.

**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.

### 3.4 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.

### 3.5 Atomicity and Durability

Like in most systems, Protea uses a write-ahead log to provide atomicity and a recovery mechanism. We assume that the log resides on stable storage which can most easily be implemented using RAID. Before any transaction executes, the first action performed is to output a beginTX record directly to storage, bypassing the system buffer. In Protea, every node possesses a write-ahead log by default. A user may turn off write-ahead logging by specifying atomic: false in the ProtoTree declaration, discussed below in section 3.6.

A user may require operator nodes to perform atomic operations (i.e., transactions). To satisfy this requirement, 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. In listing 2 is the simple idiom used to ensure that an operation is performed atomically.

```java
long tID = 0;
tID = writeAheadLog.beginTX();
writeAheadLog.appendPut(tID, key);
node.put(key, value);
writeAheadLog.commitTX(tID);
```

Listing 2: The idiom used to make an operation atomic

Currently, the system does not make use of check-pointing, so upon system crash, each log must be replayed all the way through to ensure that state is fully recovered. However, adding check-pointing as a feature would be straightforward. A naive 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.

An entire ProtoTree can be made atomic by simply declaring its root node atomic. After a system crash, the root node’s log is played back and committed transactions are re-executed on the root node. Since the log records exactly the sequence of operations performed by the root before the crash, the state of the entire tree will appear exactly as it was immediately after the last committed transaction on the log.

Which node operations must be logged in order to ensure atomicity? Gets do not need to be logged since they can only affect the system’s soft state (e.g., what is in a read cache node’s cache) and thus will not affect the way that the system state appears externally. On the other hand, both puts and deletes must be logged. For
example, suppose we issue the operation \( \text{put}(k, v) \)
on a node and later issue \( \text{delete}(k) \), after which the system crashes. If the log only records puts and not deletions, then on recovery, the node will still contain the key-value pair \((k, v)\).

The windfall of atomic operations is that logging can also provide durability for soft state, at least in that soft state can be completely reconstructed from the log during recovery if the system were to crash. For many ProtoTrees, this soft state durability property comes with atomicity by default. However, this is not the case for trees containing nodes with get methods that alter the system’s soft state. For example, a put against a read cache node may not only fetch the associated key but also migrate that key-value pair from the primary storage device to the cache device. In order to offer a durability guarantee for the soft state stored on the cache device, the read cache node must also record get operations in its write ahead log. The current implementation of atomicity in Protea does not provide durability for soft state, but adding this feature would merely entail logging get operations to the write ahead log of an atomic node.

3.6 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.

```json
{
  "type": "replicaNode",
  "id": "replicaNode1",
  "atomic": false,
  "children": [{
    "type": "logNode",
    "deviceName": "/dev/xvdb",
    "id": "hdd1",
    "deviceSize": 16000000000,
    "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 3: An example ProtoTree JSON for a ReplicaNode

3.7 Custom API’s

The current implementation is in Java. As Listing 3 illustrates, application programmers download the Protea Java package and import it where needed, then calling the `generateTree()` method of the Protea class while passing in a JSON declaration, which then returns the node at which the declared ProtoTree is rooted. The method synthesizes the ProtoTree by recursively processing the JSON declaration. The ProtoTree is constructed bottom-up, i.e. from the leaves. First the device nodes are instantiated, followed by the lowest-tier operator nodes, and finally followed by the rest of the operator nodes, ending with the root operator node.

```java
import com.github.projectdelos.Protea

Node tree = Protea.generateTree(json);

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

Listing 4: Example usage of the Java implementation of Protea

As Figure 2 shows, the end-to-end vision for Protea is as follows: application programmers import the Protea library and call the `generateTree()` method for a collection of ProtoTrees their application requires. A design decision was made here: rather than have the Protea abstraction attempt to capture all potential API’s in a
set of customizable Protean root nodes, we chose to defer that step to the application layer. Instead, Protea facilitates the construction of custom API’s by the following: each of these ProtoTrees’ root node expose the standard put-get interface, essentially giving the application programmer a set of put-get maps to use. Thus, the way that ProtoTree root nodes are used is analogous to how programmers use any other Java class. With the current Java object-oriented implementation, the programming aspect of building and configuring storage systems with a given set of hardware will already be familiar to application programmers.

4 Example ProtoTrees

4.1 Griffin

In terms of applications that utilize Protea, Griffin[6] is a good candidate. Griffin’s design is motivated by the fact that though SSD’s provide much higher performance than hard drives, they have a limited lifetime with respect to writes due to the nature of flash devices (blocks have to be erased before they can be overwritten). After a finite number of erase cycles, the SSD is no longer usable. 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 maintaining read performance, thus conserving SSD erase cycles and prolonging its lifetime. Writes are logged sequentially to the HDD and periodically migrated to the SSD in a single batch. This significantly reduces the number of erase cycles needed to serve the application. Reads are usually served from the SSD and occasionally from the slower HDD.

Griffin can easily be implemented as a ProtoTree, as shown by Listing 2. The left child of the writeCacheNode is a hard drive cache and the backing storage is a solid-state drive. Again, in the future Protea will be extended to include a more sophisticated set of cache migration schemes and support custom user built policies. The Griffin ProtoTree can be used as it stands as a simple key-value store (a natural use case might be Griffin-style network enabled storage devices). Or the application programmer can use the Griffin ProtoTree in some other custom upstream interface as an element of a larger system.

4.2 Corfu

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 main-
tains 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 Evaluation

The test shown in Figure 3, Figure 4, and Figure 5 were conducted by reading and writing to files on an SSD on a machine with a 1.8 GHz Intel Core i5 processor and 4GB of ram, so there are obvious performance overheads at work and the data is less than ideal in terms of accurately representing the performance of the system. The tests do show that Protea successfully runs end-to-end and interestingly exhibits characteristic behavior be-
tween the Corfu reads and Griffin reads. The Corfu reads perform significantly faster than Griffin reads due to the additional logic involved in the WriteCacheNode of the Griffin ProtoTree, where the cache needs to be checked for existence of data before forwarding the request to the backing storage.

We expect to see significant decreases in latency when open device files with the O_DIRECT flag is implemented in Java. Open device files in this mode allows for raw IO: operating system buffering is bypassed and bytes are sent and received directly from the device.

Figure 4 shows an evaluation of the PassThroughMapper, which is simply a pass-through map treating 64 bit keys as addresses over a BlockDevice. As a baseline, we evaluated the read and write throughput of a ProtoTree consisting of a PassThroughMapper node over a 4 MB BlockDevice node. The operations were random block-sized (i.e., 4 kB) reads and writes. The throughput for both reads and writes levels nicely, with read throughput plateauing around 220 MB/s and write throughput leveling around 52 MB/s. Currently we lack benchmark data with which to compare these figures and I/O is still being mediated by the file system, so these figures should only be taken as additional indication that the system can operate end-to-end.

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, I/O 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 interface library, giving them an interface with which to access the distributed storage space. The aspect of specialization is similar to the flexibility 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.

7 Conclusion and Future Work

7.1 Constraints on ProtoTree Declarations

As the system matures and the interface design grows more concrete, Protea must include logic to rigorously validate ProtoTree declarations. One concern that immediately comes to mind are blockSize declarations on sibling nodes. For instance, would it be valid for a replicaNode to have as children a blockDevice of blocksize 4KB and another blockDevice of blocksize 64 bytes? The initial solution seems to constrain the semantics of the system to force sibling blockDevices to have a common blockSize. In addition to having rigorously defined semantics, ProtoTree declarations could be checked for other properties such as performance. This is discussed below.

7.2 Concurrency

Concurrency control is a necessary aspect of a storage interface. As it currently stands, the Java implementation of Protea does not support concurrency control. Concurrency control can be implemented on Protea in the same way that it is implemented in other systems through the use of scheduling and locking. Given that each unique logical item of data is associated with a unique key, locks can be layered over keys. However, a complication arises due to the fact that keys are associated with logical items, not physical replicas. For instance, the key 45 might be associated with a certain value, but that value might exist across the children of multiple replicaNodes. One process could overwrite one copy of key 45 while another process reads a different copy of key 45. A plausible solution would have to distinguish between distinct physical copies of data rather than mere logical copies. One way of achieving this is to identify physical distinctness
of data through both the data’s key and the ID of the device node on which it is stored.

7.3 Cloud Integration

Protea would be well-suited for deploying on cloud computing services such as Amazon’s Elastic Compute Cloud (EC2). In addition to offering commodity computing machines, Amazon’s Elastic Block Store (EBS) provides users with persistent block level storage volumes that can be integrated with EC2 instances. Currently Amazon EBS offers three types of storage devices: general purpose SSD’s, provisioned IOPS SSD’s (maintains more than 10,000 IOPS), and magnetic storage. As hardware storage devices diversify, cloud services might offer an increasingly wider selection of storage types that Protea can take advantage of.

This notion also raises the very interesting possibility of a distributed Protea, one that is geared towards constructing distributed storage systems utilizing storage hardware on differing machines or acting as a centralized master across various network-enabled hardware. Among other issues, consistency, consensus, network partition tolerance, and availability would need to be considered in this case.

7.4 Automated ProtoTree Discovery

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.

7.5 Performance Modeling

Given the well-defined tree structure of Protea, performance can be modeled for the overall system or specific subtrees. This would not only aid the ProtoTree designer in developing efficient ProtoTrees, but paired with automated tree discovery, Protea could automatically produce optimal ProtoTrees given certain parameter constraints.

7.6 FPGA and Hardware Optimizations

Ultimately Protea is to be implemented on FPGA devices that are optimized on the hardware level for Protea-specific systems. This would provide tremendous performance speedups, as the operating system could be bypassed entirely, replaced instead by a minimal system optimized specifically for the ProtoTree at hand. The bottleneck for the application would be pushed to the network and disk I/O. The FPGA’s can then be configured similar to a traditional network-attached storage device, but instead of exposing a file system interface, Protean FPGA’s would expose the custom API’s designed by the programmer or just the default put-get interface at the root node.

8 Acknowledgements

We would like to thank Mahesh Balakrishnan for providing the vision for Protea and advising its development. His work in distributed systems and storage systems gives him unique insight and perspective for spearheading research efforts for Protea.
References


