Protea: A Framework for Building Flexible Storage Systems

Jason Kim

December 22, 2015

Abstract

Storage system design demands consideration of many different factors, including cost, throughput, durability, application workload, and availability. However, we claim that much of the complexity of storage system design can be eliminated by observing that most storage systems can be decomposed into functional modules. To this end, we present Protea, a framework for creating storage systems over heterogeneous storage hardware. We introduce the ProtoTree, a novel and powerful tree abstraction that can express a wide range of storage system designs in a simple, intuitive manner. We describe our implementation of Protea and demonstrate that it can trivially reimplement many existing storage systems with only a fraction of the time and effort. We additionally outline how Protea can be used to automate the discovery of novel storage systems over heterogeneous storage hardware.

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 a HDD, at the cost of write restrictions. The problem of designing a storage system is no longer a simple process of employing the traditional memory hierarchy. System designers must explore the complex tradeoffs between throughput, latency, read and write performance, and cost, among other considerations. This problem space is further complicated by the rapid introduction of new storage hardware. Indeed, many papers of the past decade have followed the template of proposing some novel storage system that exploits the unique characteristics of certain storage hardware in order to produce favorable properties for the entire system.

Protea provides a novel and general abstraction that can express a diverse set of existing heterogeneous storage system designs and also a vast space of hypothetical designs. This abstraction should not only allow for simpler implementations of existing systems but also facilitate the creation of novel designs.

Our design for Protea 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. Storage system designers must currently resort to haphazardly wiring together third-party or custom code that re-implements such functional idioms in order to realize novel storage designs. Protea instead offers a library of operator nodes that expose a consistent interface to the designer. An operator node is an implementation of one of the functional idioms mentioned above. Protea operator nodes expose an interface by which they can be composed together in a general yet well-defined manner.

Section 2 describes the abstraction of the ProtoTree the motivation behind it. Section 3 describes our implementation of Protea. Section 4 describes some example systems implemented in Protea. Section 5 describes Protea’s status with respect to the ACID properties. Section 6 describes our evaluation of Protea. Section 7 describes related work. Section 8 describes future directions for Protea, as well as issues and some possible solutions. Section 9 finally concludes our work and findings.

2 The ProtoTree

We represent a storage system in Protea as a tree. The leaves of a 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 view of storage systems. Our thesis is that almost any
existing storage system can be expressed as some composition of a small number of common components or idioms. Furthermore, there is an implicit hierarchy to these compositions. For instance, a write cache is a set of behaviors in software that operate on a cache device and a primary storage device. Another instance of this pattern is log-structured storage, which consists of software that operates over some storage device. Therefore a tree seems like the natural way to represent both the modularity and hierarchy of storage systems, with nodes representing storage abstractions/devices and edges representing the various possible relationships between abstractions and devices.

The ProtoTree design also encourages application programmers to think about their storage systems in a modular fashion. The design requires all relationships between various storage devices and abstractions to be explicit, making it easier for the designer to reason about their system and avoid unanticipated behavior resulting from unexpected interactions between storage abstractions.

Protea also aims to enable ACID semantics on a storage system. For instance, a designer can specify that writes to a given operator node are atomic and durable. The ProtoTree can also recover from various failure modes.

Figure 1 illustrates some relatively simple ProtoTrees. Figure 1a shows a mirroring node sitting on top of three device nodes. A write operation on the mirror node results in a write operation being issued to each of the device nodes, which in turn write to the two hard drives and one solid state drive beneath them. Figure 1b shows a write cache scheme in which a solid state drive acts as the cache and a hard drive acts as the primary storage device. Figure 1c shows a more complicated scheme in which the cache child of a write cache is another intermediate operator node. Under this scheme, data is 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.

The decision to use a tree as our central abstraction was a balance between power and simplicity. Previous work has been done to simplify the storage system design process itself. Other work [2] approaches this problem by automating the design process entirely, leaving only the task of specifying a set of requirements for the storage system, such as desired reliability, availability, and performance characteristics. This framework then attempts to automatically satisfy these requirements by computing various estimated metrics on candidate designs and selecting for designs that exhibit desirable metrics. This approach provides a useful contrast to ours and raises an important question: How much control over the system’s implementation should the designer forfeit? We make the following observations:

- For a storage system (and a system in general) to be useful, it must eventually be evaluated end-to-end according to the actual needs and workload it is intended to handle. Running a candidate system design through a predefined set of metrics can only offer a narrow, a priori evaluation of the design. This metric-based evaluation must eventually be validated by an end-to-end evaluation.

- The design framework dictates both the degree of the designer’s control over the implementation and the degree to which the resulting implementation’s structure is explicit. The aforementioned framework is pessimal in both respects. It allows for very little control over the system implementation and also completely obfuscates the resulting system’s structure from the designer.

- The consequence of the two preceding points is that any finding in an end-to-end evaluation that suggests a way to improve the system cannot be acted upon because the design framework only allows the designer to specify their requirements, not their implementation.
The aforementioned framework attempts to remove the designer from the equation as much as possible. Our preceding observations lead us to respond in the opposite manner. The design space itself must be structured and simplified so that the designer can still exert as much control over their design as possible.

Thus far, we have sketched out and motivated the ProtoTree abstraction, omitting implementation details, as well as many important design decisions. Among others, the following questions remain unanswered: What interface do nodes expose? How do different intermediate nodes interact? Where 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.

3 The Protea Implementation

Here we describe our Java implementation of Protea. First, we describe the nodes in Protea that constitute the ProtoTree, working up from the bottommost level. We describe how these nodes interface with one another, as well as how those nodes interface with storage devices.

We subsequently describe Protea’s tree generation facility, which allows users to succinctly express ProtoTrees in JavaScript Object Notation (JSON) format. The generateTree function parses these JSON descriptions into working ProtoTrees.

3.1 Device Wrappers

How does Protea enable the design of storage systems over heterogeneous storage hardware? The DeviceNode and BlockDevice classes act as wrappers around devices so that other Protea nodes can use a uniform interface to communicate with devices. This design decision separates hardware-specific device wrapper code from everything else, ensuring that the integration of new storage hardware to Protea will only entail the creation of new device wrapper classes.

Device nodes are the leaf nodes in a ProtoTree. Leaf nodes “directly” communicate with storage devices. In our current implementation, disk I/O is in fact mediated through the file system. This is undesirable due to the file system buffer cache and the added latency that it introduces. A bigger problem is that file system buffering will be invisible to Protea, meaning Protea may proceed to process new requests after receiving acknowledgement that a write to disk was successful, when in fact the file system merely wrote the update to the in-memory buffer cache. This may result in data loss in the case of a power failure. The C system call open() can be called with the flag O_DIRECT, which enables direct random access to storage disks, thus bypassing the file system and the buffer cache. Java Native Interface (JNI) offers a way to interface with native C code. We plan to eventually write JNI wrappers that will allow device nodes to interact directly with device in order to avoid the file system buffer cache.

Device files can be found in the Linux /dev directory. These device files expose a random-access, 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. In order to support block addressable devices, a BlockDevice node is placed over a DeviceNode. The BlockDevice node is initialized with a block size (e.g., 4 KB). The sole function of the BlockDevice node is to enforce block-rather than byte-sized reads and writes. Non-block-sized values are zero-padded in order to align them to block increments on disk.

3.2 Data Layouts

How does Protea determine data layout on storage devices? Either a PassThroughMapper or LogNode sits over each DeviceNode and BlockDevice. These new nodes provide two options for data layout on storage devices. We describe these nodes and their operations in more detail below:

PassThroughMapper: The PassThroughMapperNode implements a direct mapping over the address space of a block device. A key is a 64-byte address and its associated value is the byte array to be stored at the key-specified address.

LogNode: The LogNode implements a log-structured map over a device. A key is an arbitrary 64-byte string and its associated value value is a byte array. Log-structured storage benefits from sequential write speedups (see [6]), making the LogNode a useful data layout option to have on hand. We have yet to implement garbage collection for the LogNode.

As with the device wrapper classes, the admitted small set of data layout classes can be expanded to encompass other data layouts. For instance, one might want to store only encrypted data on disk. This would entail simply creating a new data layout class that performs encryption on values before handing them down to the device.

3.3 Operator Nodes

How does Protea encapsulate the functional idioms of storage system design into modular components? Protea has a library of operator nodes that each implement
one of these functional idioms. 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-byte 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. It also maintains a local hash set for determining existence of keys in the cache. Currently the WriteCacheNode implements the most naive migration policy: It migrates 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.

### 3.4 Declaring and Generating ProtoTrees

Now that we’ve described how the different components of a ProtoTree work, how do users declare and subsequently generate ProtoTrees using Protea? Users can manually declare ProtoTrees by instantiating the requisite nodes and composing them into the desired tree. This method of tree generation is cumbersome when done by hand, but it may be useful as an API for other software to integrate with Protea. It also makes it possible to create wrappers around ProtoTrees that expose arbitrary interfaces for a storage system, a usage of Protea that we explore in section 4.

The preferred method for manually declaring ProtoTrees is to write a JSON file. JSON is a very com-

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

Listing 1: Operator node interface

```json
{
  "type": "replicaNode",
  "id": "replica",
  "atomic": true,
  "children": [
    {
      "type": "passThroughMapper",
      "deviceName": "testSSD1.txt",
      "id": "ssd1",
      "deviceSize": 16000,
      "blockSize": 4000
    },
    {
      "type": "passThroughMapper",
      "deviceName": "testSSD2.txt",
      "id": "ssd2",
      "deviceSize": 16000,
      "blockSize": 4000
    }
  ]
}
```

Listing 2: An example ProtoTree JSON

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. Currently the WriteCacheNode implements the most naive migration policy: It migrates 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.

### 3.4 Declaring and Generating ProtoTrees

Now that we’ve described how the different components of a ProtoTree work, how do users declare and subsequently generate ProtoTrees using Protea? Users can manually declare ProtoTrees by instantiating the requisite nodes and composing them into the desired tree. This method of tree generation is cumbersome when done by hand, but it may be useful as an API for other software to integrate with Protea. It also makes it possible to create wrappers around ProtoTrees that expose arbitrary interfaces for a storage system, a usage of Protea that we explore in section 4.

The preferred method for manually declaring ProtoTrees is to write a JSON file. JSON is a very com-
mon data-interchange format. It allows arbitrary levels of nesting, which makes it an ideal choice for declaring recursive structures like trees. Listing 2 shows an example ProtoTree JSON file. It consists of a ReplicaNode with two PassThroughMapper children, each overlaying an SSD.

After a user has designed their ProtoTree, they can simply pass it into the generateTree() function, which parses JSON objects and generates the corresponding ProtoTrees. Since the root of any tree is simply some Protea node, the user can then use the tree by calling put(), get(), and delete() on it.

4 Example ProtoTrees

In terms of applications that utilize Protea, Griffin[7] 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 3 shows a Griffin-like ProtoTree. The left child of the writeCacheNode is a hard drive 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.

Corfu[3] 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 4 and Listing 5. 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 ACID Properties

We aim to ultimately enable the creation of ProtoTrees that perform ACID transactions over storage hardware. We discuss the constituent properties and the current status of Protea with respect to atomicity, isolation, and durability.
5.1 Atomicity and Durability

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. Listing 6 shows the simple idiom used to ensure that an operation is performed atomically.

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 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 `put(k, v)` on a node and later issue `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)`.

Listing 6: The idiom for making an operation transactional

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

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 get issued against a read cache node may not only fetch the requested value 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.

Now would be be useful to implement persistent soft state in Protea? Since replaying `get()` operations during recovery would take at least as long as processing an equal number of `get()`’s after recovery, it seems that logging them would not save any time. Additionally, `get()`’s after recovery may center around a different set of keys than those that were frequently requested before recovery, so recovering cache state might constitute a net loss of time and space. A final observation that favors not logging `get()`’s is that `get()`’s can far outnumber `put()`’s and `delete()`’s in a typical workload, so the penalty in added log size would not be insignificant.

Currently, flagging a node as atomic ensures that in the case of a whole-system failure, the system will be able recover to a consistent state. However, in the case of an aborted transaction, Protea does not ensure consistency. Ensuring atomicity even in the case of an aborted transaction is closely related to our discussion of isolation below. Suppose two concurrent transactions $t_1$ and $t_2$ are being processed and $t_2$ sees some effect that $t_1$ produced. Then $t_1$ is aborted. The system is now in a state that could be inconsistent since $t_2$ saw some state that it should not have seen. Even if we roll back all the operations that $t_1$ managed to perform, it may have influenced the effect of $t_2$ on the system. Thus, to ensure that aborted transactions behave atomically, we must hide the effects of a transaction until it has committed. To guarantee that this is upheld requires locking, which we discuss in section 5.2.
5.2 Isolation and Concurrency

Protea does not currently support concurrent operations. This is a serious impediment to its usability as a storage system framework since many storage workloads depend being able to issue concurrent reads and writes to the storage system.

As a result, we currently stipulate that users must only interface with a ProtoTree at the root node and that they must issue requests sequentially. While they can access any of the nodes within the tree just easily as the root, we cannot make any guarantees about the behavior of the system when used in this manner. For example, if a put is issued directly to a non-root node and the root node is atomic, then in the event of a system crash, recovery will not preserve the effect of this put operation since it was not seen by the root node white ahead log. Another notable example is a ProtoTree containing a mirror node over several redundant storage devices. Much of the utility of such a node is that it can improve the availability of the replicated data. However, since Protea does not currently handle concurrent operations, in order to get concurrent reads the user must risk seeing inconsistent data by reading directly from the redundant nodes. If a write operation to the mirror node in progress, then it's possible for only half of the mirror node's children to have processed the operation. Thus concurrent reads from different children may yield different values.

But what would it take for Protea to exhibit serializability? Consider two concurrent put operations issued to the root node of a ProtoTree. We cannot make any assumptions about the order in which nodes in the tree receive and process each operation other than that a node's index is modified only after it has received acknowledgement from its children that the operation has succeeded. The naive method for achieving serializability would be to implement whole-tree locking, but this would be too high a cost for many applications.

Instead, we can use locks over individual keys at each node in order to provide fully serializability, at the cost of occasional delays whenever there are simultaneous overlapping operations. When two concurrent operations attempt to access the state of a node for updating or reading, they must first obtain a lock over each key that they want to access. If they are attempting to modify the same key, locking ensures that only one of them can proceed to affect the relevant subtree, ensuring that the sub-operations of these transactions do not interleave in any potentially dangerous ways. If two transactions do not involve any of the same keys, then they can safely execute concurrently.

6 Evaluation

How does Protea perform? We conducted several tests on various ProtoTrees, measuring read and write throughput, as well as recovery time for trees that logged their transactions. These tests were conducted on a machine with a 1.8 GHz Intel Core i5 processor and 4 GB of RAM. The machine used a single SSD as a storage device. Recall that Protea currently performs disk I/O through the file system, so the numbers in this section mean little when considered in an absolute sense. Any one of a number of factors could, for instance, affect read throughput given our admittedly impoverished test setup. However, these numbers demonstrate that Protea has end-to-end functionality.
6.1 PassThroughMapper Performance

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. As shown in figure 2, read and write throughput are both extremely high (compared to typical SSD read and write throughput), with read throughput always hovering around 1 GB/s and write throughput increasing logarithmically to about 1 GB/s. This is almost certainly explained by the file system buffer cache. The buffer cache is being loaded up with the “device” blocks, causing reads and writes to only make it as far as the cache, thus exaggerating the system’s apparent throughput.

6.2 Corfu and Griffin Performance

As shown in figures 3 and 4, 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 primary storage device.

6.3 WriteAheadLog Recovery Performance

Figure 5 shows the time taken by a WriteAheadLog node to recover data to a LogNode. Once again, interestingly the recovery time scales almost exactly logarithmically with respect to the amount of data recovered. While it’s hard to determine at the moment what might be causing the logarithmic correlation, there is a positive correlation, which is what we expected.

As mentioned in section 5.1, due to the lack of checkpointing, recovery time suffers because recovery entails replaying the entire log file. Periodic checkpointing would ensure smaller log files and thus faster recovery.

7 Related Work

Other research is related to Protea in one of two primary ways. One class of research describes systems that could be implemented using Protea. Another class of research describes frameworks that share similar ideas or goals with Protea, although we have not found anything that aims to do quite what Protea does.

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

The Arrakis[5] 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.

Block Mason [?] is a block processing framework that bears conceptual resemblance to Protea. Block Mason offers a library of block processing elements that can be wired together into a graph that performs some larger, complex operation. The idea of modularizing block processing operations and composing them into more famil-
iar, higher-level operations is very akin to what Protea aims to accomplish with storage systems.

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[4] 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 erase before overwriting, making it more suitable for file system metadata which is frequently modified in small sizes. This architecture also extends the lifetime of NAND flash devices.

8 Future Work and Current Issues

We now discuss directions that we hope to take Protea in, as well as some issues that must be addressed in the current implementation in order for Protea to be a viable storage system option.

8.1 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 selection algorithms 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. In contrast with the aforementioned automated design framework in [2], Protea offers a clean interface and clear abstraction for understanding what a particular ProtoTree is doing, allowing a user to further hand-tweak an automatically discovered design.

Protea could also be implemented on FPGA devices. This could provide significant performance speedups, as the operating would be bypassed entirely. The bottleneck for the application would be pushed to network and disk I/O.

Ultimately, we see a natural extension of the single machine implementation of Protea that we currently have to a distributed implementation that can produce storage systems across many machines. Many of the common idioms of distributed storage systems can be modularized just like the idioms present in Protea now. These include the various replication schemes across machines, data sharding, configuration options for different levels of consistency, and system metadata facilities.

8.2 Issues

As our implementation of Protea currently stands, there are a few notable issues that we hope to address as development work on this framework continues. While many of these issues have been discussed in other contexts throughout the paper, it might be helpful to enumerate them all in one place.

Direct random file access: Protea does not directly perform I/O on storage hardware. Rather, I/O is mediated by the file system. This introduces a few additional layers of indirection between Protea and storage devices that it attempts to operate on. The primary issue is the buffer cache. As described in our evaluations in section 6, the buffer cache is a somewhat insidious file system optimization. While it can significantly speed up I/O, it accomplishes this by caching read and written blocks in memory. This can result in Protea thinking that an update operation was successful when in reality the file system merely cached the results in the buffer cache. This could result in data loss if the items in cache are not eventually flushed to the backing storage device.

Garbage collection: Our implementation of the LogNode does not currently garbage collect. For the purposes of implementing the first iteration of Protea, garbage collection represented a time-consuming, non-critical feature. While implementation would be straightforward and require no innovation, garbage collection is quite a chore.

Concurrency: Protea does not currently support concurrent operations. As discussed in section 5.2, key-grained locks are a straightforward way to ensure that concurrent operations over arbitrary ProtoTrees are prop-
erly isolated.

**Abort atomicity:** Protea may not fail to roll back an aborted transaction. As discussed in section 5.1, this issue, like concurrency, will be resolved with key-grained locking.

**Metadata:** The LogNode maintains an in-memory index of keys and value addresses, while the Read-CacheNode and WriteCacheNode each store an in-memory index of keys residing in the cache. Since each of these nodes is backed by a storage device, the metadata associated with them may grow to be quite substantial in size, making a naïve in-memory solution infeasible. One solution to this of course would be to bite the bullet and just swap chunks of metadata between secondary storage and memory. Another would be to allow users to specify in their ProtoTree configuration that they do not want the cache nodes to use in-memory indexes. Opting out of an index would entail that every time the cache node receives a get(), it must check the cache subtree to determine whether the key has been cached. This is slower than using an index but has the benefit of using no memory.

## 9 Conclusions

Protea is a framework for creating storage systems over heterogeneous storage hardware. Protea enables storage system design to be done in a fast, intuitive, and modular fashion, eliminating much of the legwork that would otherwise be required to hand-implement a storage system.

We demonstrated the ease with which one can use Protea to implement both existing storage system designs, as well as hypothetical designs. We then evaluated the system, demonstrating its ability to perform end-to-end.

Our work thus far shows promise in that our design seems to adapt in the right places. New hardware can be integrate by adding a simple class to Protea’s library of device wrapper classes. New data layouts can be used by similarly adding a class to Protea’s library of data layout nodes. High-level functionality can be encapsulated into a new operator node. And the interfaces defined between the different nodes of a ProtoTree ensure that Protea provides an intuitive model for conceiving of storage systems.

## References


[2] **Amiri, K., and Wilkes, J.** Automatic design of storage systems to meet availability requirements.


