

The Case for Active Block Layer Extensions*

Jorge Guerra, Luis Useche, Medha Bhadkamkar,
Ricardo Koller, and Raju Rangaswami
School of Computing and Information Sciences
Florida International University
{jguerra,luis,medha,rkoll001,raju}@cs.fiu.edu

ABSTRACT

Self-managing storage systems have recently received attention from the research community due to their promised ability of continuously adapting to best reflect high-level system goal specifications. However, this eventuality is currently being met by both conceptual and practical challenges that threaten to slow down the pace of innovation. We argue that two fundamental directions will help evolve the state of self-managing storage systems: (i) a standardized development environment for self-management extensions that also addresses ease of deployment, and (ii) a theoretical framework for reasoning about behavioral properties of individual and collective self-management extensions. We propose Active Block Layer Extensions (ABLE), an operating system infrastructure that aids the development and manages the deployed instances of self-management extensions within the storage stack. ABLE develops a theory behind block layer extensions that helps address key questions about overall storage stack behavior, data consistency, and reliability. We exemplify specific storage self-management solutions that can be built as stackable extensions using ABLE. Our initial experience with ABLE and few block layer extensions that we have been building, leads to believe that the ABLE infrastructure can substantially simplify the development and deployment of robust, self-managing, storage systems.

1. INTRODUCTION

In recent years, the opportunity of utilizing idle or cheaply available computational capability to build active intelligence into storage systems has been explored by many researchers. Such initiatives are steps towards the ultimate goal of making storage systems *self-managing*. Self-management solutions hold the potential to simultaneously improve the functional properties and reduce the overall cost of the storage system. However, despite the present interest in building them, there exist few systems that subscribe to this philos-

*This research was supported in part by NSF Grants CNS 0747038 and IIS 0534530, and the Department of Energy Grant ER25739.

ophy. Further, the pace of innovation and development of new self-management services¹ inside the operating system is surprisingly slow.

We believe that immediate effort is required along two directions to bring self-managing storage systems into the mainstream. First, there is a need for an operating system software infrastructure that can help standardize the development and deployment of self-management services; this is critical for evolving the kernel-space development experience to an acceptable level. This infrastructure must also be capable of supporting the integration of multiple self-management services inside the operating system storage stack so that system administrators can readily deploy these solutions. Second, there is a need for theoretical foundations that will allow reasoning about individual and collective service behavior and automatically compose service stacks based on administrator-defined self-management policies. Such a theoretical foundation is essential for addressing the concerns of data consistency and reliability in storage systems that continuously learn and adapt themselves.

In this paper, we make the case for *Active Block Layer Extensions* (ABLE), an operating system infrastructure that simplifies the development and deployment of self-management services within existing storage systems. ABLE is an extension of the block layer inside the operating system and itself manages a stack of active self-management services. ABLE advocates building self-management services at the *block layer*, which exports a *logical block interface* to storage clients (e.g. file systems, virtual memory, etc.) for accessing an underlying block device. Block layer self-management services automatically inherit several desirable features such as access to both process and device context of I/O operations, a simple interface to build upon, file-system independence as well as accessibility, I/O scheduling services, and easy arbitration of storage resources. We shall explore these reasons further in Section 2.

While we advocate building self-management services at the block layer, it is important to clarify that by no means do we suggest that block layer implementations are sufficient or even appropriate for *every* optimization inside the storage stack. This is especially true for services that are strongly intertwined with the semantics of other layers. However, we do claim that a large set of storage self-management solutions can be implemented most conveniently at the block layer. Further, as we shall elaborate later, optimizations

¹To simplify exposition, we use the term *service* to denote a single self-management component within a system that may be composed of several services.

that specifically operate at the file system layer can leverage the ABLE software infrastructure in a straightforward way.

For the sake of perspective, we point out that the ABLE project is necessarily the product of experiences during the development of two block-layer self-management services [6, 38]. BORG [6] is a self-management service that automatically reorganizes on-disk data layout based on block layer access patterns to improve I/O performance. BORG identifies popular sequence of block accesses and copies them to a dedicated partition of the disk using layouts that best support their relative sequence of access. EXCES [38] is a self-management service that focuses on power conservation in storage systems. Specifically, it uses alternative low power storage devices (e.g. a compact flash storage device) as a persistent cache for frequently accessed disk blocks so that disk drives can be powered down for longer durations.

While these systems confirmed the positive effectiveness and efficiency of implementing intelligence at the block layer, in each case, the development experience was poor. Apart from the well-known pitfalls in kernel space development, the development task was additionally complicated because of the multiple control paths in I/O processing, ensuring consistency of data accessed during data movement, handling of non- page/block aligned I/Os, necessity to maintain persistent data structures, and many more, to realize what would appear as fairly simple tasks (e.g., copying a set of blocks from one drive location to another). An interesting observation, however, was that a significant portion of the non-functional complexity [10] could be encapsulated within a block layer infrastructure. Incidentally, these non-functional code fragments required deep understanding of I/O handling inside the kernel and were also the main source of bugs during development. Specifically, in the case of the BORG implementation, the significantly more complex 4069 out of the total 7168 lines of code (representing 56.8% of the code-base) could be encapsulated into ABLE infrastructure. For EXCES, these numbers were even higher - 2342 out of the total 2639 lines of code (representing 88.7%) could be factored into ABLE.

Apart from the development experience, we also noted that composing aggregate services (from a set of individual self-management services) raised a lot of questions that a system administrator may be ill-equipped to answer independently. The need for a theoretical framework that would allow modeling service behavior and analyzing the behavior of service aggregates was thus identified.

Our position is that developers and administrators who optimize storage systems currently share experiences similar to what was just described. We envision the ABLE project transforming into a concerted effort to address a critical infrastructure need. We present arguments in favor of the ABLE architectural and design decisions in the rest of this paper, elaborating on the key elements of the ABLE infrastructure as we go along. We also outline example ABLE services that we have either developed or which serve as candidates for future case studies.

2. SELF-MANAGEMENT AT THE BLOCK LAYER

Storage researchers have innovated at varied layers in the storage stack. In this section, we examine the suitability of each layer of the stack in building *self-management* solu-

tions, in terms of information availability, solution generalizability, and implementation complexity.

Virtual File System. VFS layer improvements have access to process- and file- specific access information and to the VFS page cache. Further, improvements are independent of the actual file system implementation(s). The VFS layer already supports extensibility [33, 42]. However, for storage self-management solutions, the absence of device-specific information such as device status and block-level access attributes is a critical drawback. It precludes the majority of solutions that operate based primarily on block-level information (e.g., fault-tolerance [8, 40], data layout optimizations [13, 21], etc.).

File System. Innovations inside the file system have the unique advantage of having access to both file- and block-level access attributes. File system innovation has a long history, with classic contributions such as *cylinder groups* [25], *journaling* [20], and *log structuring* [32], to more recent contributions [12, 15, 16, 19, 22, 28, 39]. Unfortunately, file system optimizations are often not generalizable due to the varied designs and implementations. Differences of on-disk data format sometimes inherently restrict the adoption of new ideas [32]. Finally, detailed block-level attributes are often hidden behind the logical block interface (e.g., a RAID or NAS device).

Block Layer. The *block layer* provides access to a logical block address spaces. The block layer may internally map the single address space to several locally attached physical devices (e.g., software RAID), or to a set of remote SAN devices. We discuss the block layer in more detail shortly.

Device Driver. Device driver layer optimizations can minimize storage stack intrusion since it fully encapsulates a physical storage device. Modern storage systems, however, often use multiple physical devices in cooperation and self-management solutions typically require a global view of the device space. This inevitably requires some block layer context. Further, solutions at this layer must deal with I/O scheduler introduced shuffling of requests and lost process context due to being accessed primarily in interrupt context. These are substantial handicaps for self-management solutions that rely on having access to accurate request attributes and request patterns.

RAID Controller. Building some form of intelligence inside RAID controller firmware, has been advocated for more than a decade [26, 40]. However, solutions this down in the storage stack are limited by the same drawbacks as driver layer solutions. Further, despite the increase in their computational capability, RAID controllers are no match for the host, and may not be able to accommodate resource hungry self-management services.

Disk Drive. Modern disk drives contain a substantial amount of computational power and researchers have argued for intelligence inside the disk drive [1, 31]. Unfortunately, innovations “inside the drive” share the drawbacks of driver-based solutions when building self-management solutions. Further, an explicit change in the disk interface would be required to access I/O context information, a proposition that is at best hard to achieve.

2.1 The Block Layer Advantage

Our position is that the block layer is the appropriate location in the storage stack for a wide range of innovations.

In the case of stand-alone storage nodes, this layer is

located between the file system and I/O scheduler layers. With more recent storage architectures, the location of block layer varies. For SAN [29] devices, the block layer encompasses the bottom layer at the initiator and the top layer of the target. For NAS [18] and object-based storage [27], this layer resides within the target’s OS, as in stand-alone storage. Block layer implementations can leverage the following unique set of advantages:

1. Access to system state. Self-management solutions rely on sensing the system state continuously for information about the workload and the storage devices. The block layer has *temporal*, *process-level*, and *block-level* attributes for each block I/O request and also provides access to device state, including failed I/O operations and device failure. Software RAID infrastructure [9, 11] operate at the block layer for precisely this reason. Prior work on semantically-smart disk systems [36] proposes techniques to additionally infer certain file-level attributes at this layer.

2. A clean slate and a simple interface. The block layer is a simple layer that maps logical block address spaces to storage devices. It exports a *device address space* consisting of *fixed size blocks* and adheres to the “block consistency contract” with upper layers which requires that a block being read must contain the exact same data that was last written to it. An advantage of a functionally simple layer is that arbitrarily complex self-management services can be built at this layer as long as they adhere to this simple interface and contract. Equally importantly, as we shall elaborate in subsequent sections, this simple interface allows us to argue about behavior and safety properties for stacked extensions.

3. File system independence. The block layer enables self-management solutions that can work with multiple heterogeneous file systems and mount-points simultaneously, requiring no changes to file system implementations. This gains importance when we take into account the variety of designs and implementations of file systems.

4. File system accessibility. A block layer development infrastructure can be easily made available to the file system instance(s) above the block layer, by which file system layer innovations can tap into block layer support infrastructure. We elaborate this further in Section 3.2.

5. I/O scheduler leverage. Self-management services can generate their own I/O traffic. The additional I/O request traffic automatically leverages scheduling services such as request merging and ordering. This frees up the developer from such generic concerns of I/O handling, allowing her to focus instead on the functional aspects of the service.

6. Control and arbitration of storage resources. A block layer infrastructure can arbitrate access to the storage space on devices. Self-management services can request and use a dedicated storage partition, distinct from file system partitions as well as the partitions of other ABLE services.

3. THE ABLE INFRASTRUCTURE

ABLE infrastructure development is motivated by two high-level goals: (i) evolve the block layer development experience and reduce developer error during the construction of self-management services; and (ii) develop a reasoning engine that will analyze individual and collective service behaviors, and automatically compose service stacks to reflect administrator-defined self-management policies. To address the first goal, ABLE must provide core primitives that simplify both basic (e.g., registering a new service into the stack,

requesting memory) and complex (e.g., block-level I/O manipulation, workload summarization) tasks that service developers must accomplish. The primitives should also include simplified interfaces to access the dynamic kernel state (e.g., device status, I/O operation status, etc.). To address the second goal, ABLE must obtain information about each service during registration, and reason about its impact on the service stack, in order to make deployment decisions based both on the system state and administrator-defined self-management policies.

3.1 ABLE Architecture and Design

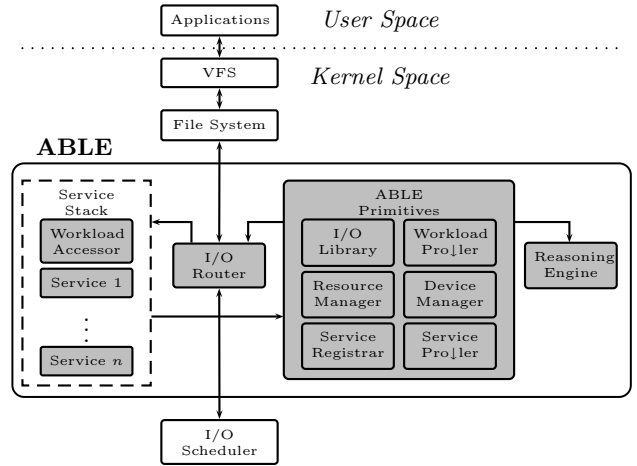


Figure 1: The ABLE architecture. Arrows represent control flow.

Figure 1 presents the architecture of ABLE, designed as an extension of the block layer in the storage stack. Our design for ABLE includes four major components. ABLE’s interaction with other layers is handled by the *I/O Router* component. Particularly, the *I/O Router* analyzes each block I/O request and routes it through the individual services in the *Service Stack* component, based on the attributes of the I/O request and service descriptions. Each service processes I/Os routed to it; such processing may include accessing *ABLE Primitives* that facilitate service development. The ABLE primitives in turn interact with the *Reasoning Engine* to manage the stack correctly.

After a service finishes processing an I/O request, it registers an I/O completion handler and returns control to the I/O router which decides the next service to route the I/O to. If none, the request continues on to the I/O scheduler. Upon completion, the I/O router invokes the I/O completion handlers of individual services in the reverse order of their invocation during the forward path, finally invoking the handler of the upper layer. Next we present a detailed description of each component.

Service Stack. The service stack constitutes the stack of the currently deployed, active, self-management services. The *Workload Accessor* service is part of the ABLE core infrastructure that is always deployed and provides input to the *Workload Profiler* sub-component of the ABLE primitives. Other services are stacked according to rules that we describe in Section 4. Each service has an input and an output *domain* of operation; the *input domain* specifies the devices and block address ranges that the service oper-

Sub-Component	Interface
I/O Library	copy_blocks, reorganize_chunk, gather_blocks, scatter_blocks
Workload Profiler	set/get_request_information, page_accessed, get_popular_pages
Resource Manager	get_service_resource_usage, alloc/free_mem, alloc/free_mem_persistent
Device Manager	get_device_characteristics, get_device_state, alloc/free_simulated_disk
Service Registrar	[un]register_service
Service Profiler	avg_service_processing_time, avg_generated_I0s

Table 1: Incomplete list of the ABLE primitives.

ates upon, while the *output domain* specifies corresponding information for outgoing block I/O requests after they are processed by the service. The domain specification of a service aids in determining service stack behavior.

I/O Router. The I/O router is the central component of ABLE that manages interaction with other layers in the storage stack as well as the internal routing of I/O requests and I/O completion events between services in the ABLE service stack. For each I/O request made by storage clients, the I/O router checks if its destination matches the input domain of individual services in the stack in top-down order. If yes, the I/O handler for the service is invoked. It is important to note that additional I/Os may be generated by a service, in which case the router will route such I/Os and I/O completion events only via the generating service and the services stacked below it.

Reasoning Engine. In a typical usage, the system administrator will use the ABLE infrastructure to deploy multiple self-management services that may be written by independent developers. Apart from choosing which services to deploy, he will specify high-level self-management policies. This component of ABLE software infrastructure provides reasoning primitives for automatically analyzing and composing service stacks to reflect administrator-defined self-management policies, based on the properties and operational domains of individual services (see Section 4).

ABLE Primitives. The ABLE primitives provide core support infrastructure for ABLE service development and administration. Table 1 lists a partial set of ABLE primitives interface, categorized based on its six sub-components. The *I/O library* facilitates service development with a set of library functions for commonly performed block I/O tasks. Some of these functions generate new I/O requests, which are handled by the I/O Router and considered as issued by the caller service. The *Workload Profiler* manages information about the requests being produced by the system and services. This data can be utilized by individual services and other components for decision making. In a similar way, the *Service Profiler* keeps track of service resource usage and other statistics, which can be used by the *Reasoning Engine* to infer changes in the system and take necessary action based on system goals, and can also be used directly for guiding manual administration.

The *Service Registrar* handles service registration and deregistration. When a request for service insertion is received by the registrar, *Reasoning Engine* primitives are invoked to determine its position in the stack. Subsequent actions en-

sure that the new service is successfully incorporated it into the system, including service variables initialization, loading of persistent metadata, and marking the service as usable. The *Resource Manager* provides primitives to allocate resources from the system. Allocations can be either persistent or not. For instance, the resource manager contains two sets of primitives that handle memory management for non-persistent and persistent memory allocation requests made by services. Synchronization policies for persistent memory in ABLE extend those proposed in the Violin project [14]. The *Device Manager* controls allocation of storage resources so services can request and independently manage storage space, both *real* and *simulated*. It also tracks information regarding the state and current usage of both physical and virtual devices. Individual services may choose to vary their behavior based on the state of a device; for example, a service may want to redirect the request to an alternate device if the hard drive is turned off or unavailable [38].

3.2 ABLE and File Systems

The ABLE infrastructure can also aid file system layer optimizations in a straightforward manner. Particularly, file system extensions can readily leverage the I/O library, workload profiler, and resource manager sub-components of the ABLE primitives to simplify their development. Requests generated due to invocations of the I/O library API are handled by the I/O router in the same fashion as I/Os generated directly within the file system. Further, complex file system optimizations can go the extra mile and isolate block-level concerns within an ABLE service. To illustrate, recent abstractions that aid reliability and performance in file systems (e.g., I/O shepherding [19], patches [15]), can largely be encapsulated as ABLE services, thus creating a generalized capability inside the OS that other file system implementations as well as other storage clients (e.g., databases, virtual memory, etc.) could then leverage.

4. A THEORY OF ABLE SERVICES

This section describes a preliminary theory of block layer services that was developed with the goal of improving our understanding of individual block layer services and their interoperation. We envision that this theory will be refined based on our experiences with applying the theory to actual implementations of block layer services and service stacks. We propose two taxonomies for classifying block layer services. The first categorization is based on *service property*, and specifies five properties for services, based on their influence on block I/O traffic:

- I. Accessor:* merely observes block I/O traffic, but does not modify it in any way.
- II. Mutator:* may modify the contents of a block, but does not change the target location of the block request or create any new block requests.
- III. Generator:* may create new block I/O requests internally but does not modify the original block requests.
- IV. Indirector:* may modify the target location of the block I/O request, but does not modify its contents.
- V. Shuffler:* may change the order of the requests issued, but does not otherwise modify requests.

The above taxonomy dictates the *class* assignment for services that satisfy a specific property. A service could, of

course, satisfy more than one property, which will lead to a multi-class assignment for the service.

The second axis of categorization is the *service goal*, which represents the primary objective of the service. Example of service goals include performance, reliability, and power conservation. These are defined by the developer, and later prioritized by the administrator.

When stacking ABLE services, the central issue is the direction of control and data flow between stacked services. Choosing an alternate stacking order may change the behavior of the storage system. Consider an accessor service that merely sends new versions of blocks to a remote logging server and a mutator service that modifies block contents. Changing the stacking order of these services changes what gets logged. A fundamental question then is: *Are there rules-of-thumb that can be used for stacking services to obtain a specified overall behavior?*

4.1 Coarse-grained Service Stacking

The *block consistency contract* requires that each block that is read through the ABLE layer contains the exact same data that was last written to it. This contract dictates safety in implementation of each class of service properties. The accessor property is safe by definition. A mutator service must ensure that each mutation of block written is a reversible; it must restore the original contents upon a read operation. A generator does not modify the original block but can issue additional write requests that change the device state; it must ensure to not overwrite blocks that are owned by upper layer entities or other ABLE services. The indirector must additionally ensure data consistency in the presence of multiple target locations for a single block. Finally, shufflers must ensure that they only change the order of requests that address different blocks.

We suggest a coarse-grained stacking order among service classes as accessor, mutator, generator, indirector and shuffler in top-down order. Since accessors do not modify requests at all, they get access to all requests, including the data of write requests first. Mutators simply mutate data without changing other attributes, justifying their relative priority over the lower layers. Generators do not modify the original block requests but may create additional I/O traffic. Indirectors are more invasive in that they may modify the target location of any block I/O request, including those created generators. Finally, shufflers must be placed last since their primary function is reordering requests issued to lower layers which must not be thwarted by a subsequent indirection operation. The I/O scheduler layer (below the block layer) is a shuffler, per ABLE terminology.

4.2 Fine-grained Service Stacking

The coarse-grained stacking theory described above is limited to services with non-overlapping classes. Further, the impact of overlapping service domains is not addressed. For accessor, mutator, and shuffler services, the input and output domains are identical.

Determining the ordering between services in the same class is complex. We propose preliminary rules for stacking based on the input and output domains of services, further guided by administrator-defined system goals such as performance, reliability, etc. Let $I(X)$ and $O(X)$ be the input and output domains of service X , and let A and B be pair of generator or indirector services. And let B be stacked on

top of A .

Rule 1: If $I(A) = O(B)$, then the service A will dominate the overall behavior.

Rule 2: If $I(A) \cap O(B) = \emptyset$, then further insight into extension B 's action is required. Thus, we consider the relation between $I(A)$ and $I(B)$. If $I(B) = I(A)$, clearly A dominates. If $I(B) \cap I(A) = \emptyset$, the extensions are independent of each other, thus no order relation exists between them. Finally, if $I(B) \cap I(A) \neq \emptyset$, A dominates.

Rule 3: If $I(A) \cap O(B) \neq \emptyset$, then A dominates for $x \in I(A)$, and as in the previous rule further analysis is needed for $x \in O(A) - I(B)$.

ABLE can use such rules to stack services to best reflect administrator-specified goals whenever the coarse-grained rules are not sufficient. For example, consider a power conservation service that caches "popular" requests to a flash device (EXCES [38]) and a performance-optimizing service that reorganizes the layout according to application access patterns (BORG [6]). If both have matching input and output domains, Rule 1 is applied and they are stacked according to the system goal defined by the administrator.

4.3 Example ABLE Services

We now briefly describe four potential ABLE services along the four axes of self-management as outlined by Kephart et al. [23] and which are substantially diverse in their operation. We have implemented two of the services described below, BORG [6] and EXCES [38].

Self-optimization: Online Block Reorganization

BORG is an ABLE service that reorganizes block data layout on a separate disk partition in their relative sequence of access, based on the observed I/O workload. This optimization lowers I/O access times by reducing rotational and seek delays. This service has three service properties: accessor, since it tracks I/O requests to deduce patterns, generator, since it creates new block I/O requests to copy data to the optimized partition; and indirector, since it may redirect I/O requests to the optimized partition.

Self-healing: Continuous Data Protection

Data loss due to disk failure and user errors (e.g., unintentional deletion) is common. An ABLE service for continuous data protection could improve failure response by making new block versions upon each block write and sending them over the network to a remote logging server, which would be accessed in the event of failure (e.g., reconstruction of a RAID drive). This service would have only access the block I/O and send some information over the network, thus it has the accessor property. It is not a generator since it does not create any local disk block requests.

Self-configuration: Adaptive Indirection

EXCES, an ABLE service, caches frequently accessed disk blocks and buffers all writes to a less power hungry device like a compact flash. It redirects block requests to the low-power device if at all possible and shutdown the disk drive during the longer idle periods, thus reducing power consumption. Similar to BORG, EXCES is an accessor, generator, and indirector.

Self-protection: Block Encryption

A block level encryption service would allow data to be accessed only by authorized users, which would solve possible security concerns regarding data privacy when a laptop is stolen, where data would be unreadable unless a disk-access password is supplied. This would have the mutator service

property, since only the contents of the request is changed.

5. RELATED WORK

The ABLE project finds motivation in several proposals on self-managing systems, notably early work on extensible and self-managing operating systems [5, 35], the HP Self-managing Storage project [2, 4], the IBM autonomic computing [23], and the CMU Self-* storage systems proposal [17]. For brevity, we employ a coarse categorization of related work as follows.

Extensible file systems. Extensible file system have been advocated since the early 90's [33]. While complementary to ABLE in general, they do share some architectural and design commonalities. Specifically, stackable vnode extensions [33] motivate stackable block layer services in ABLE, both allowing seamless extension of the storage stack. Zadok et al.'s work on WrapFS [41] is perhaps the most similar in philosophy to ABLE, where they advocate writing file system extensions as kernel modules and providing a rich support infrastructure.

FS/block-layer co-extensions. Researchers have proposed utilizing inferred or augmented FS information inside storage systems [36], and augmented block level information inside storage clients such as file systems and databases [7, 12, 34]. ABLE can aid in the development of both classes of systems, providing a complete infrastructure for developing systems of the former class as ABLE services. It can also provide partial infrastructure support to the latter class of solutions, whereby these solutions can leverage the ABLE library and resource management primitives.

Block layer virtualization. There is abundant work on block layer virtualization, each of which helps motivate an ABLE like infrastructure.

RAIDframe [9] is an early block layer framework that enabled rapid prototyping of various RAID configurations by allowing easy combination of encoding and mapping primitives. More recently, researchers have proposed automatic generation of RAID configurations based on high-level specifications of workload and deliverables [3].

Several efforts from the open source community (md [11], GEOM [37], Vinum [24] and EVMS [30]) and from industry players (HP Openview, EMC Enginuity, Symantec/Veritas Volume Manager, and NetApp FlexVol) address block layer virtualization. Typically, these systems only allow the administrator to choose from one of a few pre-existing virtualized configurations (e.g., mapping devices to a logical space and setting up RAID levels). None provide the mechanisms to create arbitrary functionality within commodity storage systems.

Of the open source initiatives, the GEOM block layer infrastructure is most similar to ABLE. GEOM classes can perform transformations on I/O requests passing through the block layer, like ABLE services. However, GEOM provides block layer development support for its classes or any libraries that simplify block layer operations. ABLE is substantially more ambitious in this regard, going beyond block layer support infrastructure to automatically composing services in the ABLE stack and providing a suite of tools that will aid both developers and system administrators.

Among academic initiatives, the recent Violin project [14] resembles ABLE in spirit. It proposes a mechanism for block layer interposition that enables storage virtualization by means of a hierarchy of extensions. While we believe

that the Violin project is a step in the right direction, it still leaves a wide chasm to cross for both the developer and the system administrator. Developers must still implement basic block-level I/O primitives themselves (such as indirection, replication, etc.), thereby leaving significant room for developer error. ABLE substantially enhances the developer experience with the many features of its core infrastructure and supporting tools. Further, Violin lacks a theory for analyzing service behavior nor does it support automation of storage stack management. Administrators must manually consider various possibilities, reason behavior, and configure their storage systems from scratch.

6. CONCLUSIONS

Self-managing stores are a critical next step in the evolution of complex, large-scale storage systems. However, there is a lack of systems infrastructure that can help enable timely development and wide deployment of such solutions. This study makes the case for ABLE, an operating system infrastructure that simplifies the development and deployment of self-management services within storage systems. ABLE attempts to address two key challenges comprehensively - developing a support infrastructure within the operating system that can speed up development and simplify deployment of block layer services, and developing a theory of block layer service behavior and interoperation. As shown with examples, ABLE can provide support for self-management extensions that improve performance, security, reliability, and power management.

Our work on ABLE has just begun. Substantial work remains to be done in better defining the scope of optimizations that can benefit from ABLE, refining the ABLE primitives, further developing the theory of block layer services, and evaluating ABLE's effectiveness in simplifying and speeding up service development and deployment. We intend to create an open-source initiative for the ABLE project that we hope will benefit from the feedback and participation of the research and open-source communities.

7. REFERENCES

- [1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. *Proc. ACM ASPLOS*, October 1998.
- [2] G. A. Alvarez *et al.* Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.
- [3] E. Anderson *et al.* Selecting RAID Levels for Disk Arrays. *Proc. Conference on File and Storage Technologies*, Jan 02.
- [4] E. Anderson *et al.* Hippodrome: Running Circles Around Storage Administration. *USENIX Conference on File and Storage Technologies*, January 2002.
- [5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. *Proc. fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, December 1995.
- [6] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, and R. Rangaswami. BORG: Block-reORGanization and Self-optimization in

- Storage Systems. Technical Report TR-2007-07-01, Florida International University, July 2007.
- [7] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high-performance parallel storage device with strong recovery guarantees, 1992.
- [8] P. M. Chen and E. K. Lee. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2), June 1994.
- [9] W. V. Courtright *et al.* RAIDframe: Rapid Prototyping for Disk Arrays. *SIGMETRICS Perform. Eval. Rev.*, 24(1):268–269, 1996.
- [10] A. M. Davis. *Software Requirements: Objects, Functions, and States*. Prentice-Hall, Inc., 1993.
- [11] M. de Icaza, I. Molnar, and G. Oxman. Kernel korner: The new linux raid code. *Linux Journal*, 1997(44es), 1997.
- [12] T. E. Denehy, A. C. Arpacı-Dusseau, and R. H. Arpacı-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. *Proc. USENIX Technical Conference*, June 2002.
- [13] R. English and A. Stepanov. Loge: A Self-Organizing Disk Controller. *USENIX Technical conference*, Jan 1992.
- [14] M. D. Flouris and A. Bilas. Violin: A Framework For Extensible Block-level Storage. *Proc. IEEE Conference on Mass Storage Systems and Technologies*, April 2005.
- [15] C. Frost *et al.* Generalized File System Dependencies. *Proc. of ACM Symposium on Operating System Principles*, pages 307–320, October 2007.
- [16] G. R. Ganger and M. F. Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *Proc. USENIX Technical Conference*, 1997.
- [17] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* Storage: Brick-based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.
- [18] G. A. Gibson and R. Van Meter. Network Attached Storage Architecture. *Communications of the ACM*, 43, Nov 2000.
- [19] H. S. Gunawi *et al.* Improving File System Reliability with I/O Shepherd. *Proc. 21st ACM Symposium on Operating Systems Principles*, Oct 2007.
- [20] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. *Proc. 11th ACM Symposium on Operating System Principles*, November 1987.
- [21] W. W. Hsu, A. J. Smith, and H. C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.
- [22] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proc. 20th ACM Symposium on Operating Systems Principles*, Oct 2005.
- [23] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.
- [24] G. Lehey. The Vinum Volume Manager. *Proc. USENIX Technical Conference (FREENIX Track)*, June 1999.
- [25] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems* 2, 3:181–197, August 1984.
- [26] J. Menon and J. Courtney. The Architecture of a Fault-Tolerant Cached Raid Controller. *Proc. of 20th International Symposium on Computer Architecture*, May 1993.
- [27] M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, August 2003.
- [28] E. B. Nightingale *et al.* Rethink the Sync. *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov 2006.
- [29] B. Phillips. Have Storage Area Networks Come of Age? *Computer*, 31(7):10–12, July 1998.
- [30] S. L. Pratt. EVMS: A Common Framework for Volume Management. *Proc. Ottawa Linux Symposium*, June 2002.
- [31] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. *VLDB*, Aug 98.
- [32] M. Rosenblum and J. Ousterhout. The Design And Implementation of a Log-Structured File System. *Proc. 13th Symposium on Operating System Principles*, October 1991.
- [33] D. S. H. Rosenthal. Evolving the Vnode Interface. *Proc. Summer USENIX Conference*, pages 107–118, 1990.
- [34] J. Schindler *et al.* Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proc. USENIX Conference on File and Storage Technologies*, March 2004.
- [35] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. 6th Workshop on Hot Topics in Operating Systems*, May 1997.
- [36] M. Sivathanu *et al.* Semantically-Smart Disk Systems. *Proc. USENIX Symposium on File and Storage Technologies*, March 2003.
- [37] The FreeBSD Documentation Project. *FreeBSD Handbook - Chapter 19: GEOM: Modular Disk Transformation Framework*. 2007.
- [38] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami. EXCES: EXternal Caching in Energy Saving Storage Systems. *Proc. IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, February 2008.
- [39] S. Weil *et al.* Ceph: A Scalable, High-Performance Distributed File System. *Proc. USENIX Conference on Operating Systems Design and Implementation*, November 2006.
- [40] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proc. Symposium on Operating System Principles*, 1995.
- [41] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. *Proc. USENIX Technical Conference*, June 1999.
- [42] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proc. USENIX Technical Conference*, June 2000.