Context-Aware I/O: Exploiting Application Context in the Storage Stack

Gopalan Sivathanu, Swaminathan Sundararaman, Kiron Vijayasankar,
Chaitanya Yalamanchili, and Erez Zadok
Stony Brook University

Abstract

We propose the concept of Context-Aware I/O (CAIO), a generic mechanism that enables lower layers of the storage stack such as the disk, to track *application-data* and *application-I/O* relationships. In CAIO, higher-level application context is propagated along with every I/O operation, in an end-to-end fashion, across the storage stack. By decoupling the generation of such contexts at the higher layers from how they are used by the lower layers, CAIO provides a simple, yet effective mechanism to encode and propagate application semantics to the storage stack.

In addition to conveying information about the logical task on behalf of which an I/O is executed, context also acts a vehicle for tracking application-specific semantics at any layer of the storage stack. A large class of such semantics can be learned implicitly by the layers, while others can be explicitly associated by way of out-of-band attributes. To demonstrate the usefulness of CAIO, we have designed and evaluated three case-studies that make use of logical contexts to track different kinds of semantic knowledge, achieving interesting functionality.

1 Introduction

The knowledge about working sets of data used by each higher-level application is quite useful in the lower layers of the storage stack such as the disk. For example, if a RAID system is aware of the data items belonging to a particular application, it can try to co-locate all data within the same disk or prefetch them to a faster storage, for power-saving and performance purposes [31, 38, 39]. From a systems-management perspective, identifying the set of hardware components containing the working set of a critical application is useful to perform operations such as selective recovery of failed hardware [24]. Disconnected operations in mobile environments [19, 20] can be achieved by proactively prefetching the working sets of running applications, at any layer.

In the modern storage stack, working set information is blurred or even completely unavailable in the lower layers. Virtualization layers such as RAID, logical volume managers, virtual machine monitors, or even a network can exist in today's storage stack, making it hard to preserve application-data relationships across layers. Techniques to address this general problem of the information-gap across layers have ranged from building application-extensible OSes [7, 14] and brand-new abstractions [12, 23, 25, 30], to more evolutionary approaches such as applications passing hints [9, 11, 27], applications implicitly influencing OS behavior [3, 8], and automatically inferring cross-layer information [3, 32]. However, none of the existing solutions enable conveying *application-data* and *application-I/O* relationships to the storage stack, in an end-to-end fashion (user applications to the storage hardware).

In this paper, we propose the concept of *Context-Aware I/O* (CAIO), a simple and generic way for applications to convey arbitrary information about their I/O behavior and relationships, without worrying about how the information will be used by the storage stack. In CAIO, an application-level *context* is propagated along with an I/O operation across the entire storage stack, in an end-to-end fashion. An application-level context is represented by one or more *context identifiers*. For example, a database application can have a unique identifier that it can propagate along with every I/O it generates, such that any storage layer can easily group all I/O generated by the database application.

In addition to working-set identification, application contexts also enable a new class of functionality that uses application-I/O relationships, such as easy and flexible performance isolation in large-scale distributed storage, and access-pattern aware caching and prefetching within the storage hardware.

To make CAIO a generic framework, we decouple the *generation* of application-level information from how the information is *used* within the storage stack. Most hint-based proposals to address the problem of information-gap in the past have tied these together. For example, in hint-based prefetching systems, the application provides hints of its future access, but the hints are specifically designed with prefetching in mind. The problem with such function-specific hints is that they require coordination and agreement between the layers involved. In a multi-vendor setup, such coordination translates into industry-wide consensus on the interface, a standardization process that takes years. In addition, such an approach cannot scale in an end-to-end manner to the multi-layered storage stacks that we have today.

Decoupling the generator and consumer of the context information leads to an interesting challenge: when the application could conceivably use more than one possible granularity of grouping I/O, how can it decide which

one to use while being oblivious to how the grouping is interpreted by the lower level? For example, a database application can group the I/O requests it generates based on the database user, session, transaction, or query on behalf of which the I/O is issued; but the lower layers are oblivious to the granularity of the context. To solve this issue, contexts in CAIO are *hierarchical*. With hierarchical contexts, higher layers can encode multiple granularities of grouping, and the lower layers can decide which granularity is the best for the particular functionality that they provide.

Context acts as a vehicle for tracking semantics at any layer by way of *implicit learning*. For example, a buffer cache layer can automatically correlate the blocks read by individual contexts and identify sequential and random streams. This information can be used to fine-tune buffer-cache policies. Beyond implicit learning, properties of I/O such as the QoS levels can also be tracked by contexts by way of *explicit attributes*. These attributes can be communicated among specific layers in an out-of-band fashion. For example, the QoS levels of individual contexts can be communicated to a disk scheduler directly without other layers knowing about it.

We illustrate the generality and power of the context abstraction by prototyping and evaluating three case studies. Our first case study is an automatic working set identifier, *WorkSIDE*, which operates at the block-based storage hardware layer. WorkSIDE automatically tracks the data working set required for an application context to run to completion. This working set can then be preloaded as appropriate in order to improve performance and availability, or to enable power optimizations. The second case study is a context-aware cache-placement algorithm within the disk that automatically learns which application-level contexts exhibit sequential streaming access pattern and avoids caching requests with that context. In our third case-study, we demonstrate the use of annotating contexts with attributes, by designing a context-based proportional-share disk scheduler. We show the usefulness of all three case-studies using prototype implementations we built for the Linux kernel, and evaluate various workloads.

The rest of the paper is organized as follows. In Section 2 we discuss the utility of CAIO by presenting a few potential applications. We discuss related work in Section 3. In Section 4 we present a taxonomy of the various kinds of contexts in storage. We detail how we generalize the CAIO interface and track semantics in Sections 5 and 6. In Section 7, we describe CAIO design and application support. We present our case studies in Sections 8 9, and 10, and conclude in Section 11.

2 The Utility of Context-Aware I/O

In this section we describe several usage scenarios that motivate tracking context information in the different layers of the storage stack. Many of these utilities cannot be implemented effectively without explicitly propagating application-level contexts. In Sections 8, 9, and 10, we demonstrate our implementation of the first three usage scenarios described below.

Working-set Aware Features. Identifying working sets of data for individual applications at the lower layers of the storage stack, enables interesting functionality such as application-aware prefetching [27], power-savings [38, 39], selective recovery of failed hardware [24], and improved data availability [31]. We describe our implementation of a disk-level working-set identifier and its usefulness in detail under Section 8.

Adaptive Caching and Prefetching. The efficacy of caching and prefetching depends on the ability to identify access patterns. Context can enable caching and prefetching mechanisms to adapt their policies based on access patterns. Section 9 describes our implementation of a context-aware disk-level caching mechanism.

Application-Aware Performance Isolation. Scheduling algorithms at different levels of the storage stack can leverage application-level contexts in scheduling decisions. For example, fair share disk schedulers can enforce fairness based on higher level logical tasks as against OS processes. Application-based resource isolation has been previously explored in the context of a single OS in Resource Containers [5]. Contexts can enable flexible resource isolation in an end-to-end fashion even in distributed storage.

Optimized Data Layout. File systems can use higher level contexts as hints for optimal data placement on disk. Co-locating files and directories created in the same context could be beneficial under certain scenarios to achieve better spatial locality during reads.

Improved Accounting. Context information associated with I/O operations can greatly help in I/O trace analysis. Trace analysis for resource consumption can be more accurate when it makes use of logical contexts pertaining to precise higher-level tasks. Contexts can also provide valuable hints about the dependencies of I/O operations and the causal relationships between them, for trace-based intrusion detection systems [18].

3 Related Work

The idea of tagging requests with identifiers has been explored in the context of distributed systems for performance debugging, profiling, etc. Pinpoint [10] and Magpie [6] are examples of systems in this category. Recently, Thereska et al. proposed applying a similar

idea in the context of distributed storage systems mainly for performance monitoring [33]. All these systems look at tagging requests in a causal chain with a certain identifier so that the entire *path* of a logical request (which may involve multiple physical network hops) can be tracked. Researchers have also looked at implicitly inferring this causal knowledge without explicit tagging [2, 15, 22] but it involves significant complexity compared to the explicit tagging approach. These systems only operate within the scope of one logical request and are targeted at a specific application. In contact, CAIO allows for a more general expression of application level semantics to cater to a wide variety of applications.

Previous work has also looked at conveying application-level grouping through new abstractions similar to our notion of context. Perhaps the closest to our work is the idea of Resource Containers [5], which allows applications to group requests into a resource container which is then treated as a logical principal for the purposes of resource isolation and accounting. However, similar to the systems discussed above, resource containers were also built with the specific goal of resource accounting and convey information on one specific kind of grouping.

Our work on context-aware I/O also fits into a class of other work on general solutions for bridging the information gap across system layers. Work in this area mainly belongs in three categories: extensible systems, hint-based interfaces, and implicit techniques to infer information or exert control. We discuss each of these.

Extensible systems. A common way to bridge the information gap between applications and the system layers is to enable the system component to be dynamically extensible by the application. Extensible operating systems [7, 29] are examples of this approach. By safe execution of application code, the operating system could allow the application to implement its own policies for traditional operating system tasks. The notion of extensibility has also been explored at the hardware level. For example, active disks [1, 28] enable applications to download code into the disk that is run within the disk controller. Such code can implement arbitrary filtering of data based on application level predicates, and even perform more sophisticated operations such as search [21] without actually transferring data out of the disk subsystem.

All these systems provide a lot of control to the application and in the process, essentially ties the layers together. Although valuable in certain scenarios, applications need to have a reasonably intricate understanding of the system in order to use these, thus making them complex to design.

Hint-based interfaces. Another approach that has been explored to solve the information gap problem is a more evolutionary one; provide specific primitives at the system level that the applications can use to convey information to the operating system. Informed prefetching [34] is an example of such a system. By enabling the application to convey information about its future access pattern, the OS acquires knowledge about the application which is used to perform more intelligent prefetching. Logical disks [11], which provides an interface for the applications to encode locality hints by creating lists of blocks, is another example. Researchers have also looked at the flip-side of the problem: provide information about the OS to the application so that the application can make intelligent decisions. Infokernel [4], and icTCP [16] are examples in this category.

One commonality between many of these hint-based approaches is that the hints are often tied to a specific kind of functionality. In other words, the information being transferred is designed with a particular purpose in mind. This in turn limits the flexibility of such a system because each new class of functionality may require yet another new primitive to be added to the interface.

Inference-based systems. The final class of related work pertains to approaches that take the extreme viewpoint along the axis of being evolutionary and being less intrusive. These systems attempt to achieve cross-layer awareness, but without explicitly communicating it from one layer to another. Gray-box systems [3] is an early example of such an approach. An application with "gray-box" knowledge of the operating system attempts to implicitly control the operating system behavior by tuning its workload in such a way that it takes the operating system to a state that results in the desired policy. Another system built along the same philosophy is semantically-smart disks [31] in which the storage system infers knowledge about the higher layers by carefully observing traffic and correlating them to higher level operations.

While being valuable from the viewpoint of being easily deployable and less intrusive, these approaches have their own limitations because they are heavily constrained in terms of not changing interfaces. This in many cases results in additional complexity making it hard to reason about correctness while also limiting the usage of such inferred knowledge to less aggressive applications that can tolerate inaccuracy.

4 Context Types

Context in storage is quite useful as seen from the kind of functionality it enables (described in Section 2). We now define *context* as follows: *A context in storage is a reference or identification used to group, on some basis, several I/O operations or data*.

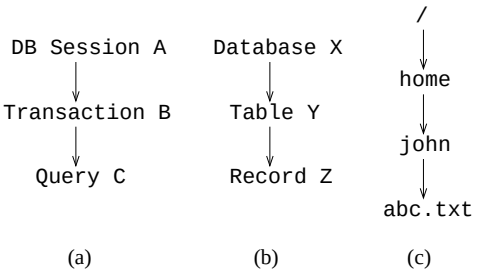


Figure 1: Examples of how hierarchical contexts can be constructed. (a) shows an access-bound context hierarchy. (b) and (c) show data-bound context hierarchies.

We now describe the types of contexts that are relevant to storage.

4.1 Data-bound vs. Access-bound

The two primary entities in storage are (a) data, and (b) I/O operations on data. Context in storage is mainly used for grouping several such data items or I/O operations. Therefore we classify context in storage broadly into two types: data-bound and access bound.

A context is said to be *data-bound* if it can be used to group several data items stored on disk, based on some metric. This grouping is independent of the way the data is accessed. For example, a data-bound context can group all blocks belonging to the same database table or file. Data-bound contexts can group data based on arbitrary criteria such as logical abstractions (files, directories, database tables, etc.), owning application or user, security domains, and so on. Data-bound contexts can be used to communicate higher-level data-structures to the disk, and enable functionality such as fault-isolated placement in RAID [31].

Access-bound contexts relate operations rather than the data pertaining to them. For example, an access-bound context can group all block write operations resulting from a single database query. Access-bound contexts enable new functionality that solely depend on the characteristics of individual I/O requests. The caching and prefetching functionality described in Section 2 requires access-bound contexts.

Figure 1 shows a few examples of context hierarchies. Figure 1(a) shows a possible access-bound hierarchy for a database application. Figures 1(b) and 1(c) show data-bound context hierarchies that communicate data abstractions.

4.2 Repeatable vs. Non-Repeatable

The lifetime of a context identifier is defined by the application that generates it. When a single context identifier is used every time to refer to a particular logical context, we call it a *repeatable* context. For example, when a context is used to group files within an access-control domain, the same identifier has to be reused every time when operations are performed on that domain. Applications have to generate such contexts using a deterministic method and may maintain persistent states to track contexts.

Non-repeatable contexts have transient identifiers. For example, if a `pid` is used as a context identifier to group I/O operations generated by a particular program, every time the program runs, the identifier becomes different, although the logical context remains the same. Non-repeatable contexts do not require any state to be maintained at the application-level.

5 Generalizing the Interface

In this section, we describe how we can cope with arbitrary context generation process at the application-level, and achieve independence between the generation and usage of application-contexts. We also describe how lower layers of the storage stack can extend contexts or correlate across different context types.

Hierarchical Contexts. To achieve generality in the CAIO interface, the context generation process at the application-level must not make any assumptions about how the lower layers use the context. However, at the application-level, there may be several different ways to generate a context, each useful for different kinds of functionality at the lower layers. A single application-wide context identifier can be used to easily group all data required by the application, whereas more fine-grained context identifiers within an application help communicate different streams of I/O requests generated by sub-components of within same application. For example, a single DBMS-wide context can be used to group all I/O and data that the DBMS manages. This enables functionality such as working-set identification for the entire DBMS. On the other hand, a per database session-level context can be used for easy performance isolation between database user sessions. We use the term *context granularity* to refer to the different possible ways to generate contexts within an application.

Therefore, for generalizing the interface without hampering the kind of functionality it enables, we evolve a context scheme where the application can encode all possible granularities as a single context, passing down *context hierarchies* (for access-bound and data-bound) rather than a single identifier. For example, a DBMS can generate access-bound contexts in granularities such as sessions, transactions, and individual queries, and data-bound contexts in granularities such as databases, tables, and records.

Lower layers of the storage stack can use hierarchical contexts without making assumptions about what each of the levels in the hierarchy mean. For example, a caching layer that wants to classify some context to exclude caching (e.g., sequential contexts) can track the

statistics on sequentiality at each level of the context hierarchy, and then choose the highest level that exhibits homogeneity in the access pattern. Depending on the specific behavior the layer is looking at (e.g., sequentiality, correlated access of the same pieces of data), the definition of homogeneity changes. Hierarchical contexts enable decoupling the application from worrying about which behavioral properties the lower layers are interested in; instead the application just conveys its state, and the lower layers make their independent decisions on the notion of homogeneity they care about, based on the layers' own per-context statistics.

Note that for a context hierarchy chain in CAIO to be meaningful, every context in the chain should qualify a logical subset of the access or data domain qualified by its parent context. For example, a per-query context identifier can be a child of the transaction identifier in which the query is a part. However, a context identifier that qualifies the class of *all* `select` queries in a DBMS cannot be a child of any particular transaction identifier, as `select` queries can be part of any transaction.

6 Context: A Vehicle for Semantics

Context in storage can also be viewed as a vehicle for tracking layer-specific semantics in the storage stack. Such semantics can be associated with contexts, by two methods: (a) implicit learning at the individual layers, and (b) annotating contexts with explicit attributes. We discuss these two methods below in detail.

Implicit Learning. Layers of the storage stack that propagate contexts can automatically learn key information about these contexts. For example, a caching layer can analyze the request stream for a particular context and classify it as sequential, random, or looping. Such information can be useful in implementing interesting policies and optimizations. In sections 8 and 9, we describe two case-studies that demonstrate the usefulness of automatic learning of semantics based on contexts. Our first case-study, WorkSIDE, learns correlations between data-bound and access-bound contexts to enumerate the working-set of data used by a particular context. In our second case-study, we implement an intelligent on-disk cache layer that learns access-patterns associated with contexts to tune caching policies.

Explicit Attributes. Certain functionality may require application-specific information to be associated with contexts. For example, a context-based proportional-share disk scheduler needs share proportions to be associated with levels in the context hierarchy. For this purpose *out-of-band* mechanisms such as `ioctl`s can be used to annotate context identifiers with functionality specific information. Note that these annotations need not be part of the CAIO infrastructure, but can be done separately between any two layers that needs to coordinate to implement a specific functionality. We describe the design and implementation of a context-based proportional-share disk scheduler that uses explicit attributes, in Section 10.

7 CAIO Design

End-to-end association of context with I/O requires passing application-generated context with every I/O operation throughout its lifetime. We evolve a framework through which context can be passed from an application all the way down to the storage hardware (e.g., a disk). In this section, we describe the changes required to the storage stack and user applications, to support contexts.

We propagate context in the storage stack by means of *context objects*. A context object contains upto two context chains, one each for data-bound and access-bound types. These context types are based on the discussion under Section 4. Context objects also carry information about the repeatability of the context chains. Repeatability is at the granularity of an entire chain and not the individual context identifiers within a chain. The structure of a context object is shown in Listing 1.

```
struct caio_context {
    int data_bound[MAX_DATA_LEVELS];
    int access_bound[MAX_ACCESS_LEVELS];
    short data_levels;
    short access_levels;
    int flags;
};
```

Listing 1: Structure of a context object. The fields data_levels and access_levels indicate the number of levels in the data and access-bound context chains. Flags contain information about repeatability and inheritance properties (Section 7.1) for the context.

7.1 Associating Contexts With I/O

The CAIO framework contains a user library that exports routines to construct context objects and add new levels of hierarchy to existing context objects. User applications can generate context objects through these routines and associate them with I/O operations. Our framework provides three different ways for user applications to associate contexts with I/O operations. They are, (a) an extended system call interface (b) group contexts and (c) context inheritance. We detail each of these mechanisms below.

An Extended System Call Interface. We have an extended system call interface that passes context objects along with storage primitives such as `open`, `read`, `write`, `unlink`, etc. Each of these I/O system calls include an additional argument for the context object. List-

ing 2 shows an usage scenario for the extended system call interface.

Group Contexts. For applications that need to perform a several I/O operations with a single context object, we provide a new system call for setting and unsetting contexts into the kernel. The scope of this association is just the specific thread of execution. Therefore applications can first set a context and then issue any number of regular I/O system calls (such as `open` or `read`), and the corresponding context object will be associated with every operation.

Context Inheritance. To support easy usage of contexts in cases where the smallest granularity is a process, our framework includes a context inheritance mechanism using which any process can set an *inheritable context* into the kernel. All child processes and threads of such a process will then inherit the same context hierarchy.

```
int fd; char buf[128];
struct caio_context *context;

/* Allocates and sets top-level databound
 * and accessbound identifers as 1 */
context = caio_create_context(1, 1);

/* Adds a new level to the access/data
 * hierarchy with identifier 2 */
caio_add_level(context, 2, 2);

/* CAIO system call interface */
fd = caio_open("/home/joe/abc.txt",
          O_RDONLY, &context);
err = caio_read(fd, buf, 128, &context);
caio_close(fd, &context);
```

Listing 2: Passing contexts from the user-level using the CAIO extended system call interface. Note that in this case group context (described in Section 7.1) can be used as well, because a single context object is used for all calls.

7.2 Context Propagation

In CAIO, each layer receives contexts from the layer above and passes it to the layer below after using them if applicable. Note that a single operation at a particular layer could translate into multiple operations in the layers below. For example, a file create operation at the file system level could result in multiple block write requests to the device driver. Therefore it is each layer's responsibility to propagate context objects appropriately to the layer below. In cases where there are more virtualization layers such as software RAID or logical volume managers (LVMs), such layers should be aware of contexts and propagate them below. Any layer can choose to store contexts in its own structures for its needs, before passing them down.

Hardware Interface Extensions. To propagate contexts end-to-end, we extend storage hardware interfaces to pass generic context objects along with every I/O request. For example, the SCSI/IDE `read` and `write` primitives take context objects. There are a number of proposals in the past that suggest interface extensions to disk systems for communicating higher-level semantic information [11, 23, 25, 30]. We believe that the generality of the CAIO interface would make it easier for disk vendors to adopt.

Dealing with Operation coalescence. Multiple logically independent I/O operations may be coalesced into one at any layer in the stack. For example, multiple file write operations to the contents of the same file block could result in a single block I/O at the disk level due to write buffering. To handle such cases, we support multiple context objects to be associated with a single lower level I/O. Layers that receive these contexts must process them one by one as if they were from different I/O operations.

Storing Contexts. Repeatable contexts may need to be stored by layers to implement optimizations that involve tracking context history, or correlating different context types. We developed a `context-store` in-memory data-structure as part of our framework to enable easy storage of context hierarchies at any layer of the storage stack. A context store manages context hierarchy in a tree structure in which each node represents a context identifier of a specific level in the hierarchy identified by its depth in the tree. Each tree node also includes a *private data* field where information about that specific chain can be stored. The context-store structure provides primitives for common operations such as adding new chains and updating private data.

7.3 Linux Implementation

We implemented our CAIO framework in the Linux kernel 2.6.15. We added new system calls for context-aware file I/O operations and implemented a user-level library for applications to easily use the new system call interface. The new system calls allowed context objects to be passed with `open`, `read`, `write`, `pread`, `pwrite`, `close`, `mkdir`, `unlink`, `rmdir` and `readdir` operations. We modified the following objects to add a new field to store contexts. (a) `task_struct` which represents a running process or thread. (b) `buffer_head` which represents a block buffer in memory. (c) `bio` which represents an I/O to a block device. The `buffer_head` and `bio` objects can optionally contain a list of contexts during operation coalescence.

We implemented the new system calls as wrappers to

the unmodified system call handlers for the operations. The wrapper system calls set the context object in the `current` task object before calling the unmodified handlers. Note that the wrapper calls unset the context upon completion of a system call, so that the scope of a passed context would be just that system call. The different layers in the OS that service the I/O operation use the context object from the `current` task object and propagate it to the corresponding `buffer_head` and `bio` objects appropriately. As the `task_struct` object is unique to a particular process or thread, this method works for multi-process workloads as well.

For group contexts, we added a new system call which assigns or removes the corresponding context in the current `task_struct` object. For inheritable contexts, we modified the `fork` system call to copy the context object of the parent, to the forked process. We also implemented the context-store data-structure as part of the kernel so that any layer such as the file system or device driver can maintain its own store.

Overall, the modifications required to implement the CAIO framework were small. We added only 350 lines of new kernel code and 150 lines of user-level code.

7.4 Application Support

The method of generating contexts at the application-level depends on specific application architectures. In general, if an application can classify its activities into distinct logical tasks, and (or) if it can group data it uses based on some criteria, it can generate contexts in a meaningful manner. Based on the kind of application, the granularity and type of contexts it can generate can vary. Some low level applications such as Unix utilities (e.g., `ls`, `cat`, etc.) can just provide an interface to the caller to pass contexts (e,g., command line arguments). We have modified some basic utility programs such as `cp`, `cat`, and `ls` to accept contexts as command line arguments. This enables a higher level caller application (e.g., a shell script) to group all its operations under the same context.

Context-Aware MySql. We have modified the MySql DBMS [26] with InnoDB [17] as the storage engine, to generate and propagate contexts at various granularities. MySql has the notion of database client connections which can obtain service from the DBMS. Each client connection gets serviced by a separate MySql thread, and can run several transactions and queries. We modified MySql to pass contexts at three granularities in the form of a hierarchy: connection-level, transaction-level, and a single query-level. Overall the modifications required to propagate contexts across the various layers of MySql and InnoDB were simple. We added only 30 lines of new code and modified 345 lines of existing code, mostly for passing an additional argument for a number of functions. We use our Context-Aware MySql as an application to evaluate our framework and some of the case-studies described in Sections 8 and 9.

7.5 Evaluation

We evaluated the overheads associated with passing context objects across the storage stack for all file system operations. In this section we first describe our test setup and the details of the experiments we ran. Note that the setup described in this section applies to all our benchmarks presented under Sections 8 and 9 as well.

We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB, 10Krpm, Ultra-320 SCSI disk. We used Fedora Core 6, running a Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean.

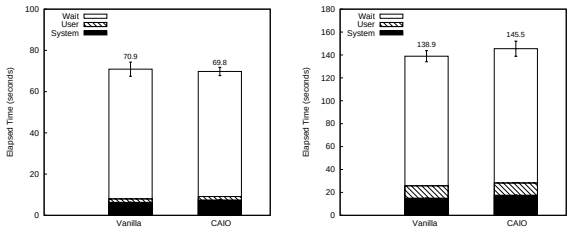
7.5.1 Experiments

In this section we describe the set of experiments and their configurations that we used for evaluating the CAIO and the case-studies.

Postmark. For an I/O-intensive workload, we used Postmark [37], a popular file system benchmarking tool. Postmark stresses the file system by performing a series of file system operations such as directory lookups, creations, and deletions on small files.

TPC-C. TPC-C [35] is an On-Line Transaction Processing (OLTP) benchmark that performs small 4 KB random reads and writes. Two-thirds of the I/Os are reads. We set up TPC-C with 50 warehouses and 20 clients. We compare our context-aware MySql running on our CAIO framework with regular MySql running on a vanilla kernel. The metric for evaluating TPC-C performance is the number of transactions completed per minute (tpmC). We report tpmC numbers for each benchmark.

7.5.2 Results



(a) 50,000 Operations (b) 100,000 Operations

Figure 2: Postmark Results for CAIO Framework

	Regular Response Time (s)	CAIO Response Time (s)
Delivery	0.096	0.109
New Order	0.039	0.064
Order Status	0.033	0.29
Payment	0.000	0.000
Stock Level	0.169	0.524
Throughput (tpmC)	67.13	64.35

Table 1: Average response time for TPC-C Benchmark

Figure 2 shows the overheads of our CAIO framework for Postmark for two different number of operations. As seen from the figure the overall elapsed time overheads were small (2% to 4%) compared to regular I/O. This overhead is mainly because of the additional user-to-kernel copies for communicating context objects from applications.

TPC-C Results. The TPC benchmark results for regular MySQL and our modified context-aware MySQL ran over the CAIO kernel is shown in Table 1. The workload loads tables into a Mysql server at start-up and runs a mix of queries on these tables for a user defined time. We configured the benchmark to run with five warehouses and created two client connection which ran queries on all five warehouses for ten minutes. As seen from throughput and response time numbers, overheads of the CAIO framework is quite small.

8 Case Study: WorkSIDE

Our first case study is the automatic **Work**ing **S**et **IDE**ntifier (*WorkSIDE*). WorkSIDE that uses both access-bound and data-bound contexts to automatically infer the minimum set of data items required to be available in order for an application (or a specific instance of an application) to run to completion. This ability to accurately identify working sets of application contexts at a fine grained level has various kinds of applications.

Performance. The working set of the application can be preloaded into a much faster but smaller memory hierarchy (e.g., a flash storage layer that provides about 100x better random access read performance), thus essentially shielding the application from performance variability due to disk access.

Availability. WorkSIDE enables fault-isolated placement of application working sets enabling truly graceful degradation during multiple disk failures similar to D-GRAID [31]. While D-GRAID could just co-locate files or directories, WorkSIDE can co-locate higher-level application working-sets within failure domains.

Power Savings. Many recent systems have looked at saving power by switching off a subset of disks in a large RAID array in such a way that applications can still function properly without the switched-off disks [38, 39]. These systems go to great complexity to identify the subset of data that is currently under use, yet these techniques are most often approximate and too coarse-grained. Being more informed about the application's access patterns and data abstractions, WorkSIDE can do a better job at such power optimizations by being more aggressive and more accurate.

Disconnected operation. Another usage scenario for WorkSIDE is when the user wants to preload the working set for a specific application context in local storage for disconnected operation, say, in a mobile environment. This enables Coda-like hoarding [19], but can be much more accurate, fine-grained and automated. For example, if the user works only on a specific build target in a large body of source code, just the subset of source files (and the metadata) needed for the target can be automatically preloaded to local storage.

The key to WorkSIDE is its ability to correlate a repeatable access context with the data context it accesses. WorkSIDE achieves this by associating with each node of the access context hierarchy, the aggregated set of data items that are accessed by that context. Semantic aggregation of such data is possible because data-bound contexts are hierarchical in nature conveying data abstractions in several granularities (such as files or directories). Tracking working set at an aggregated level enables much simpler and reliable tracking of repeatability. For instance, if an application touches different parts of a file in its different runs, block-level tracking may not find much of a repeatability, whereas tracking at the file-level would indicate the pattern. Since the data context hierarchy essentially contains information of the entire data abstraction tree, it can track this information at various granularities, and decide on which granularity provides the best trade-off between the amount of data to be preloaded and ensuring completeness for the application.

8.1 Design

To determine the working set of a higher level logical task, WorkSIDE has to track history of both data-bound and access-bound contexts for every task. We designed WorkSIDE as an on-disk mechanism to demonstrate its working as part of the firmware of a high-end block-based RAID storage system. WorkSIDE can potentially exist at any layer of the storage stack such as the file system or the device driver. Through our design, we show that even in the lowest layer of the storage stack (the storage hardware), working set identification can be done to an acceptable level of accuracy, through context-aware I/O.

For WorkSIDE to correctly determine the working set of data for a given access-bound context, the higher application has to pass data contexts to communicate the

semantic organization of data. This can relate to on-disk structures such as B-trees, database tables, files, and directories. In this section, we first detail how access-bound contexts can be associated with corresponding data-bound contexts. We then discuss a few policies that can be adopted to determine the granularity of the working set of a given context. Lastly, we present our prototype implementation of WorkSIDE.

8.1.1 Associating Access with Data

WorkSIDE maintains two context stores (described in Section 7) to track access-bound and data-bound contexts respectively. Each store has context trees to represent the hierarchy. We call tree nodes in the access and data stores as *Access-Context Nodes* (ACNs) and *Data Context Nodes* (DCNs) respectively. Note that, as data-bound context is mainly used to communicate the semantic structure of data, it need not necessarily be passed by the higher-level application for every I/O request. For example, if a DBMS uses the `table` and `record` abstractions as data-bound contexts, it may pass the context hierarchy only when such abstractions are created (e.g., a table creation) or updated (e.g., a new record insertion). For example, the DBMS need not pass data-bound contexts for every `select` query. To handle this condition, WorkSIDE may have to map access-bound contexts accompanying a block I/O request with a pre-existing data-bound context hierarchy.

The following are the contents of a DCN: (a) A context identifier. (b) The number of blocks in the entire sub-tree with the node as root. (c) A list of block numbers associated with the context (if it is a leaf node). Every time a block I/O has an accompanying data-bound context chain, the corresponding block number is added to the leaf DCN of the chain. (d) A list of pointers to its child nodes. (e) A back-pointer to its parent node. This is used to increment the number of blocks in every parent along the chain when there is a new addition to a leaf node.

While adding a node to the tree, we enforce the *single parent* constraint, where every node must have at most one parent. When there is a context chain passed, that violates this condition, we truncate the chain after the spurious node while adding it to the tree. In almost all common cases, this would not affect the accuracy of the data-bound context tree, as most data-abstractions already follow this rule. For example, a single block cannot belong to more than one file (except in rare cases such as hardlinks in Ext2).

WorkSIDE also maintains a hash table, BDTABLE, to map block numbers to the corresponding leaf nodes in the data context tree. The BDTABLE is used to lookup the data context for any block when an I/O request to it does not have an associated data-bound context. Upon receiving a block I/O request with a access-bound context, WorkSIDE can map the corresponding block number to any level of abstraction in the data-bound hierarchy by just traversing through the parent back-pointers in each node in the data context tree.

In the next section, we describe how this infrastructure is augmented with association policies to determine the optimal granularity of associating a data-bound working set for a given access-bound context.

8.1.2 Working Set Identification

Identifying the working set for a given node in the access-bound context tree involves associating that ACN with one or more DCNs. Therefore every ACN in the access store contains pointers to one or more DCNs.

Greatest-Common-Prefix Mode. We designed WorkSIDE to operate under two different modes for choosing the appropriate DCN for a given ACN. In the first (and simple) mode, which we call the *Greatest Common Prefix* (GCP) mode, WorkSIDE maintains utmost one DCN per ACN. Whenever there is an I/O in the context of an ACN, the request block number is looked up in the BDTABLE to find the leaf DCN to which the block number is associated. The leaf DCN is associated with the ACN if the ACN did not previously have a DCN associated. If not, the greatest common prefix node in the tree (starting from the root) for the new leaf DCN and the previously associated DCN is computed (using the parent back-pointers) and associated with the ACN. The working-set is enumerated by just traversing the sub-tree starting from the associated DCN. This method of enumerating the working set for an ACN ensures completeness, but under some scenarios there could be a significant number of falsely associated blocks. For example, if an access context A reads files `/home/john/plan.txt` and `/home/john/private/list.txt`, the GCP method of association would include the entire contents of `/home/john/` in the working set of A. A variant of the GCP mode mitigates this problem under some scenarios by tracking the longest depth to traverse while enumerating blocks, along with the ACN. With this, the working set of A would just include files up to depth level 3 (`/home/john/private`).

Multi-DCN Mode. In the second mode, which we call the *Multi-DCN mode*, WorkSIDE tracks a list of DCNs per ACN. Every ACN has a list of duplicate eliminated pointers to parent DCNs. To enumerate the working set for a given ACN, the following procedure is used: for each DCN associated, all blocks belonging to their immediate children are included. For example, if an ACN B reads files `/home/john/plan.txt` and `/home/john/private/list.txt`, DCNs for

/home/john and /home/john/private will be associated with B. While enumerating the working set of B, all files (not sub-directories) under /home/john and /home/john/private will be included. Therefore, the multi-DCN mode of association provides more accurate identification of working sets. However, this method needs to track more information per ACN. In the procedure described above, we choose the hierarchy one level above the leaf DCN for every block access. However, the number of such levels can be configurable based on specific system and workload requirements.

WorkSIDE can also track information required for both GCP and multi-DCN modes simultaneously (every ACN can have both the list of parent DCNs and a single GCP node). Based on the kind of usage scenario for the working set, enumeration process can be decided dynamically to choose the optimal granularity.

8.2 Prefetching for Power Savings

We developed an on-disk prefetching tool that uses WorkSIDE to enumerate the working set of access-bound contexts and prefetch them into a faster storage. For prefetching, we tracked the repeatability of the working set of each ACN, and for repeatable ACNs, we prefetch and serve the entire working set from the faster storage medium.

To evaluate our working-set aware prefetcher, we compiled several modules in the Linux kernel source, and e2fsprogs package [36], with inheritable contexts. We found that once working-sets were identified by WorkSIDE and prefetched into RAM by our prefetcher, there were no requests sent to the disk during the compile workload. Therefore, working-set aware prefetching of data enables turning off disk drives (and hence save power) in the case of repeatable workloads.

8.2.1 Implementation

We implemented a prototype of WorkSIDE and our prefetching tool as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. The pseudo-device driver receives all block requests, and redirects the common read and write requests to the lower level device driver, after storing context information that needs to be tracked. Our prototype of WorkSIDE included both the GCP and multi-DCN modes of associating data-bound contexts. It contains 3020 lines of new kernel code.

For testing WorkSIDE, we also modified the VFS layer of the Linux kernel to encode the pathname of the entity being operated (file or directory) along with every lower level I/O request. File system meta-data blocks such as super blocks, bitmaps and directory blocks have to be dealt with separately, as they may not particularly belong to a specific application. To handle such blocks, we modified the Linux Ext2 file system to associate a generic "common" context which can be interpreted by any layer as one that is not associated with any particular access-bound context. We call our modified Ext2 file system, Ext2C.

Module	# Directories	# Files	# Blocks (4k)
Ext2	14	315	1149
Ext3	14	328	1452
ReiserFS	14	328	1432
NTFS	14	320	1769

Table 2: Compilation Working Set Statistics

8.3 Evaluation

We evaluated the correctness and performance of our prototype implementation of WorkSIDE. For correctness we used a Linux kernel module build process, and for performance, we used the Postmark benchmark described under Section 7.

Kernel Modules Build. Our goal during this test was to evaluate the correctness of the working set identification mechanism of WorkSIDE. We untarred a vanilla Linux 2.6.15 kernel on our Ext2C file system mounted over our WorkSIDE pseudo-device driver. We did this through a shell that has an inheritable access-bound context set (described under Section 7.1), with depth one. We then remounted the file system to eliminate cache effects and compiled the source-code of a few file systems (Ext2, Ext3, Reiserfs, and Ntfs) under the fs/ subdirectory of the kernel source. While compiling each file system, we used different shells with different second-level inheritable contexts set. We initialized the build process through "make install" separately at the beginning, and remounted the file system after each compilation. We ran this test over WorkSIDE for both GCP and multi-DCN modes of operation.

Under the GCP mode, we noticed that the working sets of every single file system compilation was identified as a the root of the kernel source tree. This is because, a file system module compilation would refer to files under include/ and fs/ and hence the greatest common prefix node becomes the root of the kernel source.

When we ran the test under the multi-DCN mode, we saw WorkSIDE identify separate working sets for each of the file system compilation contexts. Table 2 shows the total number of directories, files, and blocks associated with the working set of each compilation. We identified these by dumping the entire access-bound context tree of WorkSIDE and their associated DCNs. In each compilation context, the generated object files were also included in the working set as the same inheritable context was passed for write operations as well.

We also used the Multi-DCN mode of WorkSIDE

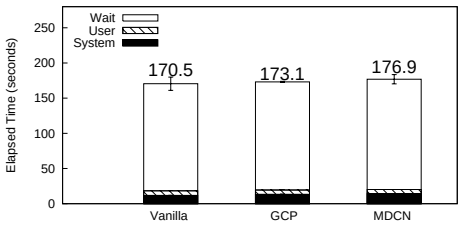


Figure 3: Postmark Results for WorkSIDE (200 Sub-directories, 20,000 Files, and 200,000 Transactions.). This shows the overheads associated with the process of working-set identification at the disk-level.

to calculate the working-sets for kernel compilation with `make allnoconfig` and `make allyesconfig`. For compilation using *make allnoconfig*, the size of the working-set came out to 32.6MB. For `make allyesconfig`, the working-set size was 3GB. As the object files during compilation are created from the same context, they were included in the working-set.

Postmark. To evaluate the performance overheads of WorkSIDE, we used an I/O-intensive benchmark, Postmark. We ran our modified Postmark that passes context objects with each I/O request, over WorkSIDE in its two modes, and compared it with regular Postmark running on top of a normal disk. For the regular Postmark we used unmodified Ext2 as the file system and for WorkSIDE evaluation, we used our modified Ext2C file system. Figure 3 shows the overheads of WorkSIDE compared to regular disks.

WorkSIDE under the GCP mode of operation had an elapsed time overhead of 1.5% compared to regular disk. The overhead mainly consists of system time (12%) caused because of updating context trees and tracking greatest common prefixes. Under the multi-DCN mode of operation the elapsed time overhead was 3.7% compared to a regular disk, caused by a 20% increase in system time. The increase in overheads compared to GCP mode is because under the multi-DCN mode, WorkSIDE has to track multiple data nodes per access-node. If WorkSIDE is implemented in a real disk, tracking context trees would be done by the disk firmware and hence would not incur the host CPU overheads.

9 Case Study: CA-Cache

Modern large-scale storage systems have hundreds of gigabytes of built-in main memory [13], primarily for caching purposes. However, today's storage systems cannot adapt their caching policies based on application-level workloads or data semantics, as they lack information about higher level semantics. This is caused by an excessively simple disk interface [12, 31]. Application-aware caching policies have been found to be quite useful in the context of OS level caches [9]. Yet today's disk systems cannot even separate independent I/O streams generated by two different applications, making it harder to implement application-aware caching policies.

In this section we design and evaluate *Context-Aware Cache* (CA-*cache*), an on-disk caching mechanism that differentiates independent I/O streams using logical contexts and tunes its caching policies based on individual access patterns.

9.1 Design

We designed CA-*cache* as an on-disk LRU write-through cache layer. The goal of CA-*cache* is to identify sequential streams of I/O and disable caching their data, as mostly sequential I/O streams do not benefit from read caching. As we are interested in the access-patterns to tune the caching policy, this application uses access-bound contexts.

Architecture. CA-*cache* consists of a set of dynamically-built context trees and an LRU cache. Each tree represents a group of hierarchical contexts with the same root context. Each node represents the hierarchical context specified by the path from the root of the tree to that node. Context trees are created or updated on each read request that specifies an access-bound context.

Classification of Contexts. Each node in the tree contains the following information about a particular context: (a) the inferred access-pattern for the particular context, (b) the block number for the last read I/O request required to track sequentiality, and (c) two counters that track the number of successive sequential and random read requests in the past. A context node is initialized as random-access upon creation. Based on the last read request and the current request, either the sequential or the random counter is incremented and the other is reset. When the values of the counters exceed a *threshold*, the node is classified as sequential or random as appropriate. Note that an already classified node could be re-classified when its access pattern changes. Upon receiving any read request, the counters in all nodes that are part of the current context are updated and the nodes are re-classified if needed. We call the number of sequential read request required for classifying a node as sequential, the *sequential threshold*. The sequential threshold is configurable, and can range somewhere between 10 and 100. A sequential-access node is re-classified as random upon a single out-of-order read.

Caching Methodology. Our classification scheme allows for different hierarchy levels in the same context chain to be classified differently. For example, two sub-contexts that are part of the same parent may be doing sequential I/O in their own levels. However, since the I/O from the sub-contexts could be received interleaved, the parent would be classified as random. CA-*cache* does not

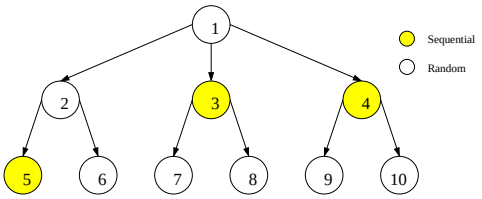


Figure 4: Context tree used for CA-cache micro-benchmark. After our micro-benchmark, CA-cache classified the grayed nodes as sequential and the rest as random.

require context identifiers to be repeatable. Therefore, it contains a mechanism to automatically forget contexts based on a timeout. We periodically purge context tree entries that represent inactive contexts (without any requests) beyond a time threshold.

9.2 Implementation

We implemented a prototype of our on-disk caching mechanism as a pseudo-device driver in the Linux 2.6.15 kernel similar to WorkSIDE. We maintain the context trees in memory and an asynchronous kernel thread wakes up periodically to purge timed out context entries. If the block is present in the LRU cache, the pseudo-device driver services the request from the cache, thereby avoiding a request to the lower level. Otherwise, the request is directed to the lower level and the cache is updated on completion of the request, if the request belongs to a random-access context.

Read Micro-benchmark. To evaluate CA-*cache*, we ran a micro-benchmark that generates synthetic random and sequential read workloads simultaneously and calculated the overall throughput of the random workload. We compared the throughput results of CA-*cache* with a vanilla LRU cache layer which treats all contexts equally. Both CA-*cache* and vanilla LRU cache used 4MB of cache (1,024 4KB pages) for this benchmark.

We ran a user program that generates workloads shown in Figure 4. The user program has four execution contexts (threads), A, B, C, and D which use their own files for I/O. Thread A reads a 4GB file sequentially with context {1-2-5} (see Figure 4). Thread B reads a 4GB file sequentially, but it uses contexts {1-3-7} and {1-3-8} for alternate reads. Thread C is identical to thread B, but it uses contexts {1-4-9} and {1-4-10}. Thread D reads random locations from a 4GB file using context {1-2-6}. For thread D, we use a random number generator that repeats itself every 1,024 reads. The threads run until any one of the sequential threads exits after reading 4GB of data. In our experiment, the throughput of the random workload when run under the vanilla LRU cache was 0.098 MB per second, whereas with CA-*cache*, the throughput was 7.71 MB/Sec.

MySQL Micro-benchmark. For this benchmark, We created two identical tables SEQ and RAND in MySQL with 4,200,000 records each, and ran random and sequential query logs simultaneously. The tables were approximately 233MB in size. The sequential query log contained a `select *` query on the table. For a random workload, we selected a subset of the records at random and issued select queries based on their record IDs. To show the benefits of caching random streams alone, we repeated the random query log ten times. We also ran the sequential log in a loop till the random workload completed. We determined the throughput of the random workload (number of queries executed per second) while the sequential workload was running in parallel. It was 266.13 queries per second without selective caching, while it was 614.15 queries per second with selective caching.

10 Case-study: CA-schedule

To demonstrate the usage of contexts in scenarios that require application-specific information, we designed a proportional-share disk scheduler, CA-schedule, that uses logical contexts to identify tasks and insulate their performance. Individual contexts are annotated by their resource share allocations. These annotations are made though an out-of-band channel between the applications and the disk scheduler.

Design. CA-schedule is a time-slice based proportional-share disk scheduler that uses resource share allocations associated with individual logical contexts, to make scheduling decisions. The share values for each logical context has to be preset by an offline communication channel between the applications and the disk scheduler. CA-schedule decides the next I/O request to be scheduled, based on the share proportion assigned for the particular logical context and the time each context has consumed. CA-schedule ensures that a particular context is given the proportion of disk-time which is *at least* equal to its share value, provided the context has enough I/O traffic to make use of it.

CA-schedule maintains several request queues based on the granularity of proportions needed. If the minimum granularity of a proportion is $1/n$, then CA-schedule maintains n request queues. CA-schedule uses equal time-slices to service each of these n queues. However, based on the share value of a context, its requests are striped across several queues. For example, if the value of n is 10, and the share level for context A is $1/2$ then the requests from that context are striped across five different queues exclusively. If we assume there are five other contexts each with share level $1/10$, this mechanism will ensure that the requests of context A will be serviced five times for every single time any other context is serviced.

	Run 1 (Ops)	Run 2 (Ops)	Run 3 (Ops)
context A (1/8)	Null	193k (144k)	142k (108k)
context B (5/8)	861k (861k)	728k (718k)	540k (538k)
context C (2/8)	Null	Null	298k (215k)

Table 3: Read micro-benchmark for CA-schedule: Each column in the table presents the total number of 4KB reads performed in a five minute interval. The values specified in braces is the ideal number of reads that should have been performed based on the share-level for that context. Each row indicates a particular context run in parallel with other contexts in that column. A "Null" value in a column indicates that the process for that context was not run in parallel.

Implementation. We implemented CA-schedule by modifying the existing fair-share scheduler in the Linux kernel version 2.6.15, popularly called as Complete Fair Queuing (CFQ) scheduler. We modified CFQ to perform proportional-share scheduling by having a constant number of queues based on the value of n and to stripe requests on the corresponding queues, based on associated contexts. The modification was quite simple: we modified 40 lines of existing code and added 120 lines of new code to the scheduler.

Evaluation. To verify the correctness of CA-schedule, we ran a read workload. We assumed three different contexts A, B, and C for all tests with share values $1/8$, $5/8$, and $2/8$ respectively. We ran 10 identical processes for each context, each performing several 4KB random reads on their own file. We calculated the total number of reads completed by the threads of each context every minute. Using this, we measured the percentage slowdown experienced by each context as other contexts were run in parallel and compared it with each context's share allocation.

Table 3 shows the number of operations completed by each of the contexts when run together with other contexts. Overall the total the slowdown experienced by each context is proportional to the share allocation.

11 Conclusions

As Butler Lampson said, interface design is one of the most complex aspects of system design, while also being the most important. Interface designers have traditionally embraced the philosophy of minimalism—hide as much information about the layers as possible, so that the layers can innovate and evolve independently. This approach, despite all its merits, has the downside of obscuring what a layer knows about its inputs, thus limiting functionality. At the other extreme, some systems have explored how to completely tie the layers together, by having extensible layers or exposing detailed information about the inner semantics of a layer. What we proposed in this paper is a middle-ground where we send a small amount of information across layers, but by making the generation of the information separate from how it is used, we enable the layers to be independent of each other while still enabling arbitrary grouping information to be conveyed across the storage stack. We have shown through two case studies that contexts are a simple and general abstraction to convey application information.

Future Work. We plan to explore more applications of CAIO, especially those that require annotating context identifiers with semantic information (at any specific layer) by offline methods as described in Section 5.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, CA, October 1998. ACM.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Bolton Landing, NY, October 2003. ACM SIGOPS.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001. ACM.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing, NY, October 2003. ACM SIGOPS.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 45–58, New Orleans, LA, February 1999. ACM SIGOPS.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modeling and performance-aware systems. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 85–90, Lihue, Hawaii, May 2003. USENIX Association.

[7] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[8] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the Annual USENIX Technical Conference*, pages 29–44, Monterey, CA, June 2002. USENIX Association.

[9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[10] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 595–604, Bethesda, MD, June 2002. IEEE Computer Society.

[11] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Prin-*

ciples (SOSP '03), Bolton Landing, NY, October 2003. ACM SIGOPS.

[12] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[13] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems. Product description guide, 1999.

[14] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[15] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 395–408, San Francisco, CA, December 2004. ACM SIGOPS.

[16] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 317–332, San Francisco, CA, December 2004. ACM SIGOPS.

[17] InnoDB. Innobase oy. www.innodb.com, 2007.

[18] S. King and P. Chen. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM SIGOPS.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[20] G. H. Kuenning, G. J. Popek, and P. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. In *Proceedings of the Summer 1994 USENIX Conference*, pages 291–303, June 1994.

[21] L. Huston and R. Sukthankar and R. Wickremesinghe and M. Satyanarayanan and G. R. Ganger and E. Riedel and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 73–86, San Francisco, CA, March/April 2004. USENIX Association.

[22] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 173–186, Berkeley, CA, USA, 2004. USENIX Association.

[23] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 105–120, San Francisco, CA, December 2004. ACM SIGOPS.

[24] K. Magoutis, M. Devarakonda, and K. Muniswamy-Reddy. Galapagos: Automatically discovering application-data relationships in networked systems. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 701–704, Munich, Germany, May 2007. IEEE.

[25] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41:84–90, August 2003. ieeexplore.ieee.org.

[26] MySQL AB. MySQL: The World's Most Popular Open Source Database. `www.mysql.org`, July 2005.

[27] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[28] E. Riedel. Active disks: Remote execution for network-attached storage. Technical Report CMU-CS-99-177, Carnegie-Mellon University, November 1999.

[29] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

[30] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[31] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[32] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[33] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06)*, pages 3–14, Saint Malo, France, June 2006. ACM.

[34] A. Tomkins, R. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114, Seattle, WA, June 1997. ACM SIGOPS.

[35] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. `www.tpc.org/tpcc`, 2004.

[36] T. Ts'o. E2fsprogs: Ext2/3/4 filesystem utilities, 2008. `http://e2fsprogs.sourceforge.net`.

[37] VERITAS Software. VERITAS file server edition performance brief: A PostMark 1.11 benchmark comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.

[38] C. Weddle, M. Oldham, J. Qian, A. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 245–260, San Jose, CA, February 2007. USENIX Association.

[39] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping Disk Arrays Sleep through the Winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 177–190, Brighton, UK, October 2005. ACM Press.