# Software defined just-in-time caching in an enterprise storage system

S. Seshadri
P. H. Muench
L. Chiu
I. Koltsidas
N. Ioannou
R. Haas
Y. Liu
M. Mei
S. Blinick

*A software defined storage environment is one in which logical storage resources and services are completely abstracted from physical storage systems. Therefore, not only can storage resources cross physical boundaries, but they can also be defined by software and provisioned automatically, for instance, by the applications that consume them. In this paper, we present a novel software defined cooperative caching (SDCC) framework that operates at the block layer and manages the placement of data in different tiers and caches that span multiple servers and storage systems in an integrated and coherent fashion. A programming interface complements the core framework by giving the applications an interface to control data organization across the storage, thereby allowing the block storage infrastructure to be software defined. The SDCC framework allows applications to actively influence the data layout while also benefitting from the system-wide knowledge and resource management capabilities of the storage system. We present an experimental study conducted using real workloads, and the results demonstrate the performance benefits gained with SDCC, as well as the potential for consolidating multiple different workloads that share the same storage server.*

## Introduction

*Software defined storage* (*SDS*) is expected to play a key role in enabling the software defined datacenter of the future, defining the evolution in the design of storage systems to enable the vision of a *software defined environment* (*SDE*). By automatically allocating workloads to the most suitable resources of the infrastructure, SDEs accelerate the deployment of workloads and optimize the use of resources, thereby delivering the agility needed to satisfy fast-changing workload demands cost-effectively. SDS is expected to accommodate architectures that can scale in terms of number of compute and storage nodes (*scale-out*) and support virtualized environments. In such environments, differentiated *service-level agreements* (*SLAs*) are achieved through software instead of hardware, and the consumers of storage can configure and use the storage resources through streamlined *application programming interfaces* (*APIs*) and policies.

Datacenters are adopting Flash in various forms. For example, introduced as a cache or as an element of tiered storage in *storage area network* (*SAN*) storage nodes, Flash causes little disruption to traditional storage operations while bringing already significant performance improvements. However, throughput increases at the storage nodes can introduce contention in the network between the host systems and storage, so distributing Flash closer to the workloads not only reduces latency but also spreads the load, enabling greater scaling. To mitigate the disruptive effects of this scale-out approach applied to traditional SAN-based storage environments, we introduce the *software defined cooperative cache (SDCC)*, a novel framework that leaves ownership of data to the SAN storage nodes even though data is stored in SAN storage nodes as well as across distributed host-side Flash. As a result, functions such as backup, mirroring and high availability operate as usual at the SAN storage nodes without impact from the distributed host-side Flash.

Similarly to software defined networking, which transfers networking control plane and possibly even data plane functions from networking hardware into software layers at the servers to achieve global optimization across the network, SDS is characterized by a horizontal dimension of global

Digital Object Identifier: 10.1147/JRD.2014.2303595

optimization and a vertical dimension of software integration. SDCC displays these two dimensions as follows: to benefit most from the low-latency access characteristics of Flash as compared to magnetic disks [1], data stored in Flash should be as close as possible to the host processors, driving the integration of Flash control and data planes into software at the hosts systems. The horizontal dimension of global optimization driven by host-side Flash comes from the coordination between hosts and SAN storage nodes: to maximize the benefits of host-side Flash, in particular as a read cache, placement of data at the distributed host-side caches and the SAN storage is continuously optimized according to workload behavior.

A further element in the vertical dimension of integration is achieved from a data and control plane perspective by enabling workloads to participate more explicitly in the optimizations involving the host-side Flash and SAN storage. To that end, a new API layer is introduced to extend the interface available to applications: on the data plane, applications provide hints that are exploited by the underlying caching mechanism, thereby managing the cache content more actively; on the control plane, applications can adjust the properties of the cache based on workload demands, SLAs, and properties of the infrastructure resources. We refer to this interface as *SDCC-API*.

The combined benefits of SDCC and the API provided result in an SDS solution that closely integrates workloads and host-side Flash while performing a global optimization with SAN storage. This can be seen as an example of software defined cache management. The SDCC layer and SDCC-API have been productized in the IBM Easy Tier* family of products.

In contrast to approaches where Flash-based caches are integrated into a distributed file system, the framework introduced here operates at the block level. As a consequence, its benefits are available to any file system or application. On the other hand, file systems access higher-level semantics of data that can be used to influence caching policies. For instance, a file system can detect sequential accesses even if they end up being spread over multiple block devices, so it can avoid "polluting" the cache (e.g. loading data unnecessarily) and instead use pre-fetch and write-behind (i.e., periodic flushes of the cache). In our case, such knowledge can be conveyed over the SDCC-API interface. Moreover, compared with other host-side block-level caches, SDCC is unique because of its cache coherency across hosts, achieved through coordination with the SAN storage system, as discussed in the Related Work Section.

An environment that can take immediate advantage of the proposed software defined cache management is server virtualization with SAN-attached storage: by adding Flash caches at the servers running the *virtual machines* (*VMs*), the performance bottleneck of SAN storage can be alleviated, latency can be substantially reduced, and hence the number of VMs can be increased, all without disrupting the storage management processes in place at the SAN storage. Depending on the virtualization technology used, the cache driver and VM-specific features may either be implemented in the physical block storage stack with appropriate hints from the layers above or be integrated in the hypervisor stack or the guest VMs themselves.

The main contributions of the paper are as follows:

- We introduce a novel framework for software defined cooperative caching in SAN environments with Flash-based *direct-attached storage (DAS)* devices on the host servers.
- We present a set of APIs for data placement, focusing on just-in-time cache data placement.
- We describe the architecture and implementation of the caching function that manages placement of data in the host server DAS tier.
- We present the results of an experimental study based on real-world hardware and workloads that demonstrate the benefits of the caching function of the proposed framework.

The remainder of the paper is organized as follows. We motivate the need for an SDS framework in Section 2 and introduce our approach. In Section 3, we provide a description of the data-placement APIs and present the architecture of SDCC. The implementation is discussed in Section 4, and the results of our experimental study are presented in Section 5. We discuss related work in Section 6 and conclude in Section 7.

## Software defined data placement

Realizing the benefits of SDEs requires that the infrastructure automatically adapt to the workload and application characteristics, system resources, and service policies. Two existing approaches at opposite ends of the spectrum address this requirement at the block-storage layer. At one end, the *config-time* approach allows the infrastructure to expose different classes of devices and allows system administrators or end users (such as application developers or applications themselves) to choose devices based on service and performance expectations at the time of configuration. At the other extreme, the *per-I/O* approach resorts to tagging each individual I/O operation with information such as the priority and performance expectations.

The config-time approach—used for example in the OpenStack** Cinder scheduler [2]—is an essential and simple starting point for achieving the end goal of SDEs. However, such config-time definitions do not allow

applications or users the flexibility to dynamically change data layouts at runtime based on user actions or system events. It is also highly likely that the workload is not well defined and may evolve over time. To be able to address the above issues, applications or users need to overprovision the system at the time of initial configuration, thus sacrificing resource utilization. Also, a config-time approach may not be appropriate for fine-grained control of runtime resources such as caches. Even if the storage layer automatically adapts to the changes in application workload characteristics, it does so reactively. As a result, the end user may suffer degraded performance until the storage system catches up with the workload.

The per-I/O approach is suitable for managing resources and providing information that is specific to an I/O request. For example, such an approach may be suitable to allow the application to influence scheduling policies to allow higher-priority requests to complete faster than lower-priority requests [3, 4]. However, this approach has its own limitations. First, the information reaches the storage layer at the same time as the I/O. As a result, the approach may not be conducive to proactive data placement, i.e., preparing the system for an impending workload. For example, such an approach may be unsuitable to specify desired data layouts such as "prefetch data into cache" or "place data on the DAS tier." Moreover, a performance overhead may be incurred because the I/O path needs to be modified. Also, the amount of space available for tagging in the context of an I/O command is limited.

In this paper, we propose a new, orthogonal approach, that is, *runtime data-placement APIs*. The goal of this API-driven approach is to allow the application developer and system administrator the flexibility to perform specific configuration and optimization actions at runtime, in response to user actions or system events. In this approach, the target layer (such as the storage system or host-side cache) exposes a set of APIs that may be invoked at runtime either programmatically (by the application developer) or through the command line (by the system administrator). Our runtime data placement APIs are designed to coexist with the config-time and per-I/O approaches, thereby providing a more diverse and flexible infrastructure over which applications can operate. When the API is invoked, the information is conveyed to the storage system through special-purpose *small computer system interface* (*SCSI*) commands; thus, it does not involve any changes to workload I/O.

We envision that the API has other broader capability aside from cache data placement, such that it can be used to communicate event-based, debugging, servicing, and policy information to the target layer and perform data placement at other layers (such as tiers of a multi-tiered storage system). However, these topics are beyond the scope of this paper.

### *Just-in-time cache data placement APIs*

The SDCC data placement APIs allow an application, platform, or similar entity (collectively referred to as *source*) to pass data placement hints to a *target* layer that implements the SDCC-API interface. There are two APIs in this framework: *Query API* and *Hint API*. The Query API allows the source to fetch information (such as metadata) from the underlying target layer. The Hint API allows the source to pass information to the target layer as follows:

```
status_descriptor Hint (
    application_ID_descriptor,
    address_descriptor,
    target_tier_descriptor,
    metadata_descriptor,
    intent_descriptor,
    in_application_priority_descriptor,
        lease_descriptor
)
```

The hint specifies a data location using an address descriptor (*address_descriptor*) to which the hint is applied. If the target layer operates at a different object granularity than the source (e.g., block vs. file), an appropriate translation layer is required to translate the address_descriptor to addresses understood by the target layer. For the cache data placement functionality described here, the *target_tier_descriptor* represents the cache. The hint may also specify an application identifier (*application_ID_descriptor*) and priority (*in_application_priority_descriptor*) to address resource-sharing issues in a multi-tenant environment, such as multiple VMs sharing a cache. In addition, the source layer may also specify an event through the *intent_descriptor* for the target layer to perform appropriate actions based on this information. For example, specifying a "close" event may result in the cache layer evicting the data in response. Similarly, specifying an "open" event may result in the cache layer pre-fetching the data. In the absence of an intent_descriptor, the target layer interprets the hint as "place data in cache." In another example, the application could use this API to specify a desired response time or throughput to a storage system implementing the API. The storage system may use its knowledge of the current performance characteristics of the system and devices to decide on an appropriate location for the data that would match the application needs.

The source layer may also specify metadata associated with the data location using the *metadata_descriptor*. Finally, a *lease_descriptor* may be used to describe the duration of validity of the hint (such as start and end times). The lease_descriptor information can be used to optimally schedule data migration, for instance,

by prioritizing data migration for hints with earlier start times.

The hint results in the target layer issuing a *request_id* and other status information as part of the *status_descriptor*. The source layer uses this request_id to track completion status of the hint through the hint management APIs, which have been omitted from this paper.

The Query API is as follows:

```
metadata_descriptor Query(
    application_ID_descriptor,
    address_descriptor,
    target_tier_descriptor,
)
```

All field interpretations are similar to the Hint API. The Query API returns a *metadata_descriptor* object with information about the current state of resources or the current location of the data.

### *Motivating example*

Consider a scenario in which a database application performs a reorganization operation to eliminate performance-degrading fragmentation in a tablespace or index. The reorganization operation involves copying over the data from the original fragmented location to a new target address space that is based on unfragmented, physically contiguous pages. However, during the reorganization operation, key performance statistics metadata associated with the source location is lost and must be re-learned by the system. Also, the cache may have been populated with data from the source address space such that switching over to the target address space at the end of the reorganization will affect performance adversely. The SDCC-API hint mechanism provides a solution to this challenge by allowing the application to proactively influence cache placement by conveying event-based knowledge to the caching layer. Once the database reorganization operation is complete, the application may first query the source metadata, issue a hint to delete the source address ranges from the cache and then another hint to promote the target address range to the cache and copy the required historical access information from source to target.

Note that any application or platform layer can directly invoke the SDCC-API. For example, in the above example, the SDCC-API calls can be initiated by the database application, by an application administrator through a command-line interface or by the underlying platform. The SDCC-API is exposed through a Java library in the current version.

## Architectural design

In this section, we present an architectural overview of SDCC. The framework implements a client-server architecture and consists of two layers:

- *SDCC-API Layer*: The SDCC-API layer provides the APIs required for the applications (and administrator) to influence data placement and caching policies. The API layer also provides hint and conflict management functions.
- *SDCC Layer*: The SDCC layer provides the cache management functions, the clustering functions required to maintain cache coherency, and the communication protocol and channel control to communicate with the storage nodes. The *SDCC client* components run on the host servers (i.e., the SDCC clients or hosts). The *SDCC server* components execute on the storage backend node attached to the SAN fabric. Each client is clustered with one or more servers. The clients communicate with the server using an in-band protocol, referred to as the *SDCC Protocol*, implemented using the SCSI protocol over the storage network fabric.

SDCC clients communicate with the SDCC server to join the cluster, to report local access statistics to the server, and to request permissions to cache data, among other things. The server manages clustering and cache permissions and provides caching advice to clients. An overview of an SDCC cluster is shown in **Figure 1**.

### *SDCC-API layer*

The SDCC-API layer intercepts hints issued by applications and communicates with file system services to translate hints from application objects into block-level addresses. The translated hints are passed on to the SDCC client that communicates with the appropriate storage node. The SDCC client may interpret hints, choosing to incorporate the information directly into its caching policy when the hint is targeted at data placement in the cache. In other situations, such as when the data placement is aimed at a tier in the storage backend, the hint is sent to the SDCC server for interpretation.

On the storage node, hints are processed by the SDCC server to devise an appropriate data placement strategy based on several additional considerations, such as scheduling, resource management and conflict resolution policies. The data placement decision is then communicated to the appropriate SDCC client as part of the SDCC server advice.

Since the SDCC-API is a feature intended to provide additional capabilities to an application, we do not provide a performance evaluation of the SDCC-APIs. Instead, we focus on the performance evaluation of the SDCC caching layer in the Section 5.

### *Role of the SDCC server*

The SDCC server plays three key roles related to hint management and advice generation: persistency, scheduling,

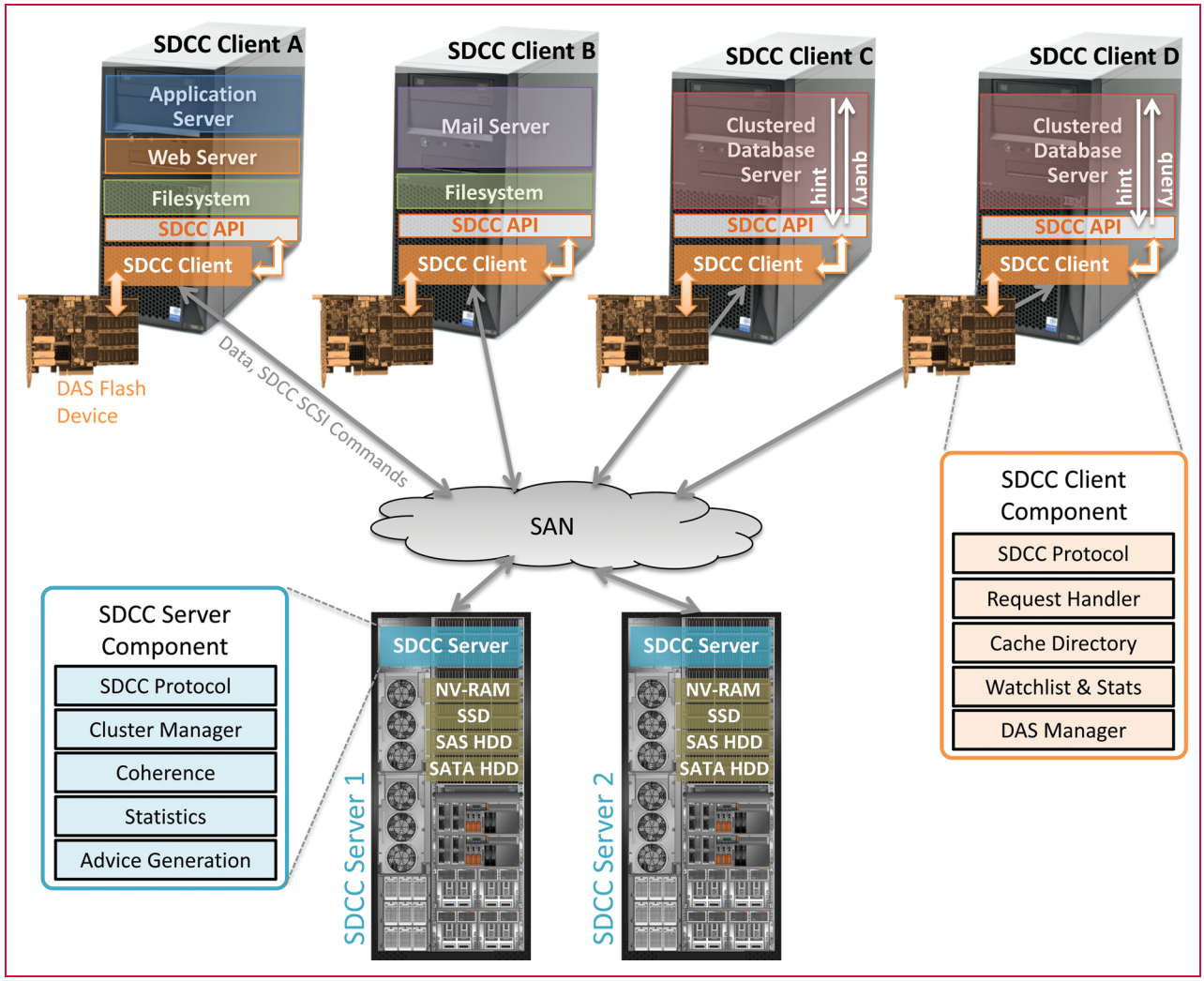


Figure 1

Architectural overview of software defined cooperative caching (SDCC). (SCSI: Small Computer System Interface; HDD: hard disk drive; SAS: serial-attached SCSI; SSD: solid state drive; SATA: serial advanced technology attachment; NV-RAM: non-volatile random access memory.)

and placement decision. The SDCC server is responsible for maintaining cache state and hint information persistently across failures and power cycles. It also manages hint lifecycles. The goal is to relieve the application from managing resources and maintaining state. For example, the application is not expected to relinquish capacity allocated in the cache or on a *solid-state drive* (*SSD*) tier for a hint after the validity of the latter has expired. Likewise, an application needs not persistently maintain state regarding previously issued hints across crashes or restarts.

An application can use hints to pre-populate a cache or tier prior to a start time when the application expects to start accessing that data. The SDCC server can schedule data migration based on available migration bandwidth, capacity available at the target location, and the "start time" for the hint. This allows the system to maximize utilization by not prematurely populating data in the target location. It also allows applications to issue hints associated with well-known scheduled tasks (such as a monthly report). The SDCC server performs data placement based on its knowledge of the workload access characteristics and system configuration in addition to application hints. Whereas each application only has a localized view of the system, the storage backend has a more global view because of its ability to monitor accesses from multiple applications and multiple hosts. While hint parameters such as the in_application_priority_descriptor may provide applications some flexibility to specify priority in a given host, the sharing of global resources such as capacity on high-performance storage requires a global view to ensure

high system utilization. The location of the SDCC server on the storage node makes it an ideal choice to combine application hints with global information to make better data placement decisions. For example, performing read caching at a host for data being frequently updated by other hosts is not efficient, as it results in frequent cache invalidates and slowdown of writes. Note that application hints arriving at the SDCC server are not blindly deemed important but are constantly evaluated. With its global view, the SDCC server can identify and place such data at a common backend location (such as an SSD tier in the backend). The SDCC server is also responsible for managing multi-tenancy issues, such as resource sharing between multiple conflicting application hints, and for enforcing policies.

### SDCC layer

The SDCC layer coordinates host-side Flash *DAS* with SAN storage to continuously optimize the system based on workload behavior and application hints. The SDCC server generates caching advice for the different SDCC clients (i.e., suggesting to each client which data are likely to become hot) and pins data to the client DAS Flash if required (i.e., forcing the clients to cache some extents). The SDCC client is responsible for managing the local DAS Flash cache space, monitoring the local workload at a fine granularity, and moving data from the SAN to the DAS Flash. For scalability reasons, workload monitoring and data placement at the SDCC server occur at a coarse granularity in terms of space and time. Here the unit of data movement is an *extent* (1 GB in size), and data placement decisions are evaluated every 24 hours for the SAN tiers to have a suitable tradeoff between the cost of data movement and workload adaptation, and every 15 minutes for the DAS cache tiers to communicate hot data suggestions and pin data to the client DAS caches in a timely manner. On the other hand, the client can afford to track data at a finer granularity: data is organized in *fragments* (1 MB in size each), and data movement decisions are evaluated continuously, on a nearly per-I/O basis. Since the SDCC server is oblivious to user accesses that result in hits in the local caches of the SDCC clients, the SDCC clients periodically report summaries of these hits to the server using the SDCC-API. In this way, the server knows the big picture about the cluster workload, albeit with some delay. On the other hand, the clients are better informed about the current workload on the host servers and can make more timely decisions. SDCC aims to combine the completeness of the information maintained at the server with the accuracy of the information maintained by the client to achieve the most effective caching on the clients.

In terms of implementation, at the storage node, the SDCC server component is part of the volume virtualization and automated tiering component. At the hosts, the SDCC client is in the block layer of the host *operating system* (*OS*) or the hypervisor, where it uses the DAS Flash to store unmodified data, similarly to a read-only cache. However, it not only caches hot data based on local application access patterns, but also allows the storage node to use the host DAS Flash as an external tier, in which the storage node can pin data for a long term when deemed appropriate. Because the storage node has a global view of the cluster workload, it can coordinate the data flow in the cluster and allow the host caches to work in a cooperative way with SAN tiering mechanisms.

The main design objective of SDCC is to accelerate host applications by taking advantage of the DAS Flash: SDCC brings the most valuable data as close to the application as possible by storing a copy of it on the DAS Flash, where it can be accessed with low latency. Moreover, by serving most of the requests from the DAS Flash, SDCC offloads significant read traffic from the SAN, reducing host and storage network infrastructure requirements and allowing the SAN storage node to serve more hosts and serve write I/Os more efficiently. Thereby, it enables the SAN storage infrastructure to scale out. It is key for the SAN storage node to retain ownership of the data, i.e., at any given time the most up-to-date version of the data is in the SAN. In this way, SDCC does not disrupt features offered by the SAN, such as high availability, copy services and tools for manageability. At the same time, SDCC is transparent to applications and file systems running on the hosts. Finally, SDCC manages the coherence and the consistency of data in the cluster, ensuring that the data stored on the DAS Flash of each host are up-to-date.

#### *Transport layer and communication protocol*

The SDCC protocol is implemented as a set of vendor-unique SCSI commands. A client initiates communication with an SDCC server by sending a SCSI command. An SDCC server initiates communication with a client by responding to a polling command that the SDCC client has sent to the server within the client I/O timeout period. The client sends another polling command when it receives a response to a previous polling command. A response to a polling command contains either the message itself or a handle that the client can use to read the message with a separate SCSI command. These polling commands are issued on an on-demand basis and are not continuous, except for the infrequent keep-alive commands that clients send at coarse intervals (e.g., every 15 minutes).

#### *Data sharing*

In many environments, multiple clients access the same volumes on a storage node. SDCC allows data sharing by multiple clients, so that clustered applications, e.g., clustered databases such as IBM DB2* pureScale* [5] and Oracle

RAC** (Real Application Clusters) [6], and clustered file systems, e.g., IBM GPFS* (General Parallel File System) [7] can be accelerated. To ensure coherence and consistency, the SDCC server hands out caching permissions to clients: a client may only cache a fragment of data (i.e., 1 MB of data) if and only if it has been granted the appropriate permissions by the server. The server will revoke caching permissions for a fragment cached on a client if some other client issues a write to that fragment. The SDCC server suppresses thrashing due to writes by not granting caching permissions for fragments that receive too many writes from multiple clients.

### *Cache coherency*

Initially, clients obtain leases from the SDCC server to become members of the cluster. With such a lease, the client can request subscriptions to one or more volumes. With a volume subscription in place, the client will periodically (e.g., at 15-minute intervals) receive caching advice from the server for that volume, which, among other things, includes a list of extents the client should promote and pin into its cache for as long as it is not advised otherwise, and a list of extents that are expected to be hot on that client and that the client should promote, or at least consider promoting, to the cache. A volume subscription allows the client to request caching permissions for specific fragments of that volume. Once this permission has been granted, the client can promote the respective fragments into its cache. At any given time, the client may relinquish caching permissions and volume subscriptions, or decide to drop its lease and leave the cluster. Similarly, the SDCC server may decide to revoke caching permissions or volume subscriptions or revoke the lease from a client if that client stops reporting its status.

### *Selective caching*

The SDCC client cache employs selective population to decide what to promote into the cache and when to do so. Therefore, the client will not cache all user reads; instead, it will carefully monitor the user workload, also taking into account the caching advice received from the SDCC server, incorporating hints coming from the SDCC-API, to only promote into the cache data fragments that are deemed of high caching utility or that the application specifically requests caching for. This is an iterative process that constantly adapts to the workload and re-evaluates the caching utility of data fragments. Note that legacy applications not employing the SDCC-API running on a SDCC client will still benefit from caching happening underneath. Selective caching achieves a three-fold objective:

a) Increase the hit rate: by only caching valuable fragments, the client avoids cache pollution with cold data, thereby saving space in the cache for more valuable fragments that will bear more hits.
b) Maximize the read bandwidth: by only writing to the DAS Flash hot fragments that are likely to stay in the cache for a long time, the write bandwidth consumption of the DAS Flash is minimized, leaving more bandwidth available for serving cache read hits at a lower latency.
c) Maximize the Flash lifetime: by keeping the rate of promotes low, the number of Flash writes is minimized, thereby increasing the DAS Flash lifetime, which is especially important with low-endurance consumer-level Flash.

## Implementation

### *Cache organization*

The SDCC client cache maintains a cache directory structure, in which allocation of DAS space occurs at fragment granularity, whereas cache hits and invalidates occur at sector granularity. The cache uses a variant of the generalized CLOCK algorithm for fragment replacement [8]. For each fragment, the cache also maintains a metric of its caching utility, which is computed based on the recency and frequency of accesses to that fragment. In addition to the cache directory, the cache maintains a shadow cache structure, called the *watchlist*, which holds information about fragments that have not yet been promoted into the cache. Fragments that have recently been accessed or have been suggested for promotion by the SDCC server are inserted into the watchlist, where their accesses are being monitored for some time. Once a fragment becomes hot in the watchlist (i.e., once it has seen a sufficient number of "pseudo-hits"), it becomes candidate for promotion into the cache. At that point, the cache directory will compare its utility against the utility of the candidate(s) for eviction from the cache and the average utility of recently evicted fragments: if the candidate for promotion is deemed more worthy of caching, it will be promoted into the cache; otherwise it will stay in the watchlist until it becomes hotter than what is to be evicted (or until it gets cold and is dropped from the watchlist altogether). In this way, the cache ensures that only the hottest data will be promoted and that they will remain cached for as long as they remain hot, avoiding unnecessary cache churn. At the same time, the cache is resistant to sequential scans and can adapt to changing workloads, as every aspect of the cache admission policy is dynamic. Besides the client cache admission logic, the SDCC server can force specific fragments to be promoted to the cache and remain there until further notice.

### *Bandwidth throttling*

SDCC operates in an asynchronous manner, i.e., without intervening in the user data path. For the SDCC client cache, this means that fragments are promoted outside of the user

read path: when the cache promotes a fragment, it first requests caching permission from the SDCC server for that fragment (if it does not have one already) and then initiates a read operation from the SAN to fetch the data. Clearly, a high promote rate (e.g., during the warm-up phase of the cache) may impact the user workload if the SAN is operating near its peak throughput. To avoid that, the SDCC server periodically informs each SDCC client about how much bandwidth can be used for promote reads based on the current storage node load. In this way, each client can limit its rate of promotes to ensure that the user workload will not be affected, even when the cache intends to promote heavily.

### *Persistent cache*

Thanks to the coordinated and cooperative nature of SDCC, the SDCC client caches can be persistent across orderly client reboots and power cycles. Upon a client shutdown, the client cache flushes its directory and relevant metadata to the DAS Flash devices, with each device storing the metadata for the data fragments it contains. Upon reboot, the cache directory can be reconstructed by reading the relevant metadata from the DAS devices that are still present. Note that during the downtime of a client, some other client may have issued writes to the data cached by the client that is down, running the risk of serving stale data from their DAS Flash upon reboot. SDCC ensures consistency by having the client inform the server about the shutdown and having the server continue monitoring the caching permissions that had been granted to that client. While the client is down, the server marks permissions for fragments that have seen writes by other clients as revoked. Upon reboot, the client is notified by the server of the fragments with still valid caching permissions: the client keeps these fragments in its cache and discards all others, thereby avoiding the risk of serving stale data. Currently, the cache contents are not maintained upon an un-orderly shutdown (e.g., a crash); the system will automatically detect the crash, however, and ensure correctness.

### *Platforms*

We have implemented SDCC starting with the high-end IBM server and storage platforms. As SDCC server we have used the DS8870 storage node [9], in which the SDCC component has been implemented within the Easy Tier automated tiering component [10]. Once the administrator enables SDCC, the system starts monitoring workloads and makes volumes eligible for caching on the DAS space of the hosts, without disruption to the high availability, manageability and reliability features of the storage system.

On the host side, the SDCC client has been implemented on the IBM Power* platform [11] as a driver for the AIX* and Linux** OSs. In both cases, the client comprises both a kernel component and a user-space component. The kernel component is in the block layer of the OS and intercepts user read and write requests. Read requests are redirected to the DAS Flash if they are cache hits. Write requests that update cached data result in cache invalidations; updates to hot data are selectively chosen and asynchronously reflected in the cache. The administrator of the host OS can selectively enable caching for some of the volumes connected to the host using a command-line interface and monitor cache operations using a command-line utility. In terms of host memory usage, the SDCC client has a small footprint, requiring only about 0.05% of the DAS size in main memory. The DAS hardware used is the IBM EXP30-Ultra SSD Drawer [12], the IBM FlashSystem* all-Flash arrays, and the PCIe** (Peripheral Component Interconnect Express) cards [13]. The SDCC-API has been implemented, but the applications we have experimented with in this paper have not yet been modified to use it actively. However, the cooperative caching at the SDCC layer has been implemented: the SDCC server will monitor the global workload and advise the clients regarding what to cache, in addition to providing coherence and consistency.

An early version of SDCC has been shipped with the DS8870 storage systems under the name Easy Tier Server. This initial version supports AIX on Power servers as the client platform with a subset of the features described herein.

## Summary of experimental results

We present an evaluation of the SDCC framework to demonstrate the effectiveness of data placement close to the application and to show that clustering and communication overheads do not interfere with performance gains. These results focus on the SDCC layer, that is, on cooperative selective caching; further performance gains will be achieved by leveraging application hints via the SDCC-API.

Since SDCC is mainly suited for transactional workloads that benefit significantly from low-latency access to data, we have evaluated it under a brokerage transactional workload and an *on-line transactional processing* (*OLTP*) workload with high skew. The former simulates a class of applications that facilitate and manage transaction-oriented business processes, resulting in a highly random I/O workload with about 90% read and 10% write operations. We used the brokerage workload to demonstrate how SDCC improves the application transaction rate. The latter is a typical OLTP workload that consists of 40% reads and 60% writes, with a highly skewed distribution in the address space: 90% of the I/O operations are contained within 5% of the address space. We used the OLTP workload to demonstrate the response time improvement under constant throughput. We have used IBM DB2 9.7 as the database server. Note that in all experiments all traditional caches, such as OS and file system caches, storage server caches, etc., operate as usual, i.e., SDCC supplements

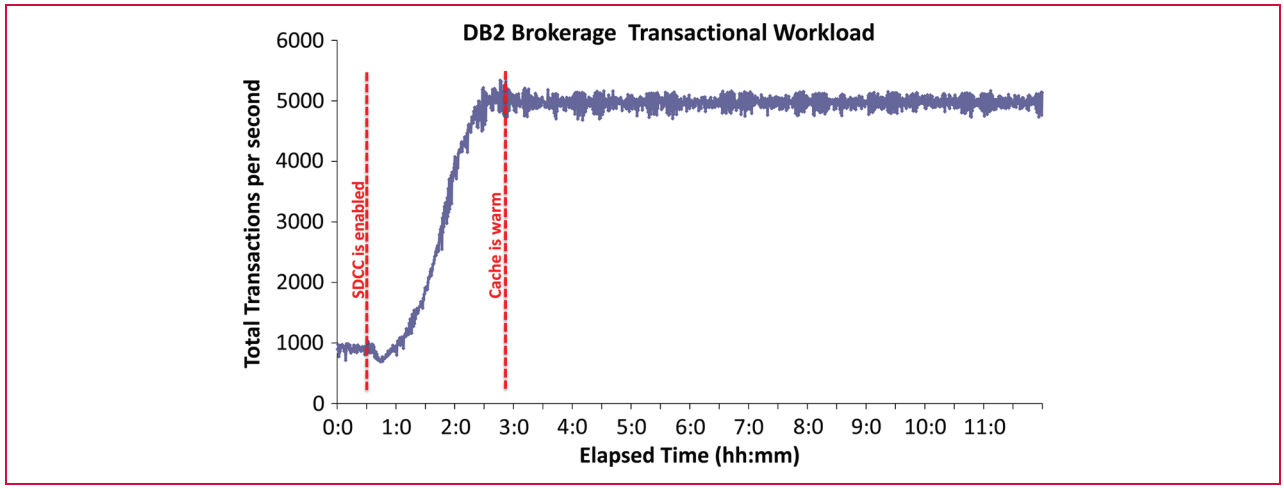


Figure 2

Transactional throughput for the brokerage workload.

**Table 1** Average response time with and without SDCC for the brokerage workload.

| | *Average Response Time (ms)* | | | | | |
|---|---|---|---|---|---|---|
| | *Customer Portfolios* | *Trade History* | *Trade Order* | *Trade Updates* | *Trade Results* | *Overall* |
| Without SDCC | 46.19 | 1507.82 | 75.99 | 2077.15 | 70.82 | 134.58 |
| With SDCC | 6.20 | 118.48 | 10.43 | 193.58 | 13.48 | 16.18 |
| *Reduction* (%) | −86.6% | −92.1% | −86.3% | −90.7% | −81.0% | −88% |

any other kind of caching at the host servers and storage systems. The hardware and software configuration we used is described in the Appendix.

The brokerage transactional workload required about 30 minutes to reach steady state before SDCC was enabled, moving the hottest of data to the client DAS Flash and resulting in an improvement of 5.4x at the steady state as shown in **Figure 2**. The first 30 minutes in the run correspond to the steady-state behavior without SDCC. At the steady state with SDCC, the client cache hit rate was about 93%, computed as a per-minute average. The 20% drop in the transaction rate immediately after SDCC was enabled is due to the very aggressive initial population of the cache, consuming read bandwidth from the SAN and impacting the user workload. During that phase, which lasted for about 15 minutes, about 200 GB of data were brought into the cache and the hit rate rose from 0% to about 40%. The bandwidth throttling mechanism described in the section "Bandwidth throttling" aims to alleviate this problem. **Table 1** shows the response time improvement for the various transaction types. Overall, the transaction response time was reduced by 88%, resulting in a significant application acceleration. In addition to these performance gains, SDCC relieved the SAN from a significant portion of load: after SDCC was enabled, and despite the throughput increase, the DS8870 served 60% fewer reads per unit of time than before. This not only allows the SAN to serve more writes, but also allows it to serve more hosts, achieving higher consolidation and substantial cost savings.

For the OLTP workload, the transactional throughput of the system was fixed throughout the experiment, which corresponds to an I/O throughput of about 40k IOPS (I/O per second). The client cache hit rate was about 80% at steady state. The results for the I/O throughput and response time are shown in **Figure 3**. Again, as data moved to the DAS Flash, the response time was reduced from 3.9 ms to 1.9 ms, a 69% improvement. Note that this improvement was achieved despite the significant write ratio of the workload (60% writes vs. 40% reads).

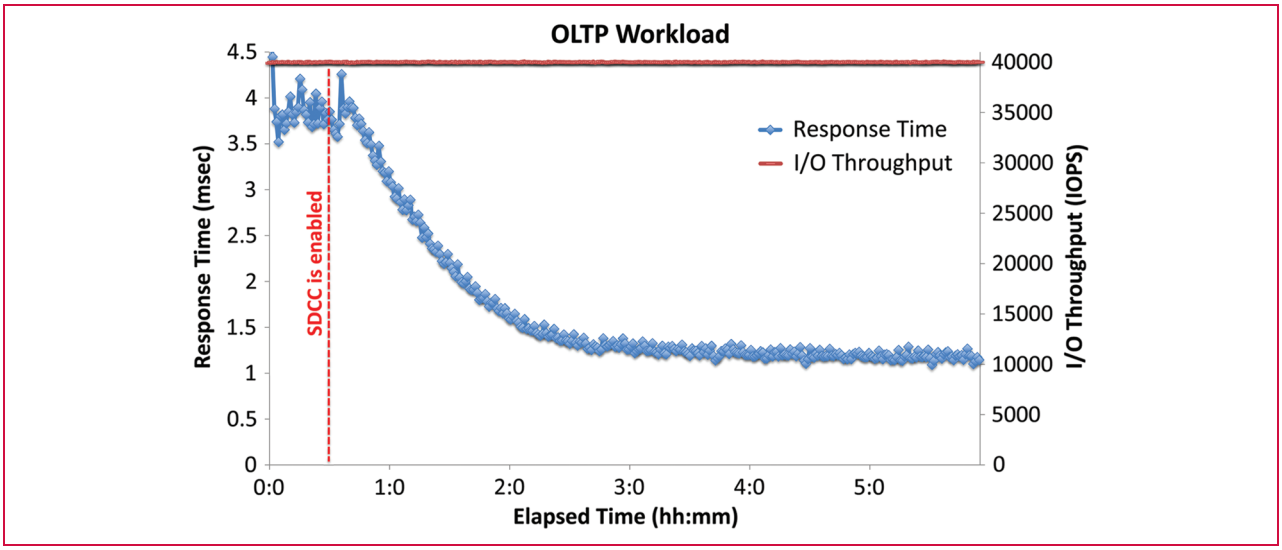


Figure 3

Input/output throughput and response time for the OLTP workload.

## Related work

The placement of Flash within the storage hierarchy is being actively researched. In [14, 15] Flash is used as a cache at the host, whereas [16] considers Flash as a cache in a storage system, and [17] as a tier in the storage system. SDCC is novel in that it integrates Flash as a cache at the host in cooperation with the storage system, providing cache coherency and cache hinting for the Flash cache at the hosts. Coherency between hosts and storage is addressed by [18] for networked file systems, whereas SDCC provides this capability for block storage systems. Cooperative caching between hosts and storage, with the host issuing demote notifications to the storage is introduced in [19], whereas SDCC provides hints for promotions and demotions along with the duration of those hints.

Many approaches to optimizing storage systems using semantic information from the host have been explored. Some approaches [20, 21] consider modifying the storage system to determine cache placement according to heuristics about the file system. This approach is susceptible to changes in the file system format that could unexpectedly cause undesired results. The SDCC-API provides direct semantic information and hence does not introduce any susceptibility to changes. In contrast to [22], the SDCC-API sends semantic information in independent commands instead of doing I/O tagging so that a diverse set of future semantics can be provided without being limited by the constraints of the widely used SCSI command set. Avoiding I/O tagging also enables the SDCC-API to provide hints for optimizations that are outside the scope of a single I/O. In addition, the SDCC-API can provide the capability to control features of the underlying storage system such as point-in-time copies.

A common industry approach to deploy host Flash caching is to combine a host device driver with SSDs or PCIe Flash, such as [23–26]. These device drivers do not maintain cache coherency across hosts, whereas SDCC adds cache coherency across hosts through coordination with the storage system. Another approach to maintaining cache coherency is to deploy Flash in combination with HBAs (host-bus adapters) [27]. The HBA solution supports caches that are hundreds of gigabytes in size, whereas SDCC supports caches that are tens of terabytes in size with gigabytes of meta-data in host DRAM (dynamic random-access memory). HBA solutions can pool resources of multiple HBAs, but accessing cached data shared among HBAs in the same host consumes SAN bandwidth [28]. SDCC shares its cache among HBAs without consuming SAN bandwidth.

## Conclusion and future work

In this paper we have focused on SDS and, in particular, on a framework that enables a software defined datacenter to take advantage of Flash to increase performance and optimize resource utilization. We have presented the SDCC-API, a runtime data placement API, that facilitates just-in-time placement of data at the most appropriate location, with an emphasis on cache placement for host-side caches. SDCC coordinates SAN storage with DAS storage to continuously optimize data placement based on workload behavior and

hints provided through the API. SDCC implements host-side caching on Flash DAS, manages the cache coherence in clustered environments in which multiple hosts access the same data, and selects the most appropriate data to be cached at each host. The framework, which we have implemented in a real system using the IBM DS8870 Storage Server and IBM Power servers, allows applications, such as the IBM DB2 Database Server, to increase their throughput by 5.4 times and reduce latency by 69% for real workloads.

In the future, we plan to further develop the SDCC-API to offer a more diverse interface and integrate it into appropriate data-intensive applications and management tools. In addition, we plan to integrate SDCC into virtual computing infrastructures, where SDCC can be used to accelerate virtualized storage and provide virtual-server-specific functions. Moreover, we plan to leverage novel types of memories (such as phase-change memory) to further improve performance and optimize resource utilization.

## Appendix

For the experimental evaluation, the results of which are presented in Section 5, we used the configurations detailed below. Here the "2 + P" designation indicates that there are three drives in total in the RAID (redundant array of independent disks) array, and a RAID stripe consists of 2 data segments and 1 Parity segment. For the brokerage workload, we used the following:

- IBM DS8870 Storage System with eight CPU cores and 256 GB cache
  - 128 146 GB 15K HDDs (hard disk drives), RAID-5
  - 4 Device Adapter Pairs
  - 4 × 8 Gb Fibre Channels
- IBM Power 770+ Server (AIX 6.1.8.0), 8 8-Core P7+ (3.8 GHz)
  - 1 EXP30 Ultra SSD
  - 2 + P SSD RAID5 arrays used as Flash cache
  - 1024 GB of DRAM
- DB2 9.7 FP2
  - 1 DB2 Instance
  - Database size was 2 TB
  - 2 1.3TB volumes were allocated for database, temporary files, and data generation
  - 1 50 GB volume was allocated for log files

The OLTP workload used the following configuration:

- IBM DS8870 Storage System with 8 CPU cores and 256 GB cache
  - 192 146 GB 15K RPM drives, RAID-5
  - 4 Device Adapter pairs
  - 4 × 8 Gb Fibre Channels
- IBM Power 770+ Server (AIX 6.1.8.0), 8 8-Core P7+ (3.8 GHz)
  - 1 EXP30 Ultra SSD
  - 2 2 + P RAID5 SSD arrays used as Flash cache.


## Acknowledgments

SDCC is the result of a collaborative effort across many IBM groups and locations around the globe. We thank all contributors, especially Timothy Brock, Chiahong Chen, Justin Cripps, Veronica Davila, Clement Dickey, Clint Hardy, Kaisar Hossain, Xiao-Yu Hu, T. Chip Jarvis, Paul Jennas, Cristoph Keil, Chao Guang Li, Maohua Lu, Joshua Martin, Jared Minch, Wolfgang Mueller, Dat T. Pham, Roman Pletka, Juan J Ruiz, Limei Shaw, Roger Strommen, Maoyun Tang, Jay Waters, Sonny Williams, Yan Xu, and Hui SH Zhang.




## References


1. E. Eleftheriou, R. Haas, J. Jelitto, M. Lantz, and H. Pozidis, "Trends in storage technologies," *IEEE Data Eng. Bull.*, vol. 33, no. 4, pp. 4–13, Dec. 2010.
2. OpenStack Foundation, OpenStack Cinder. [Online]. Available: http://wiki.openstack.org/Cinder
3. J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage performance virtualization via throughput and latency control," *ACM Trans. Storage*, vol. 2, no. 3, pp. 283–308, Aug. 2006.
4. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, X. Jian, R. Menon, and T. P. Lee, "Performance virtualization for large-scale storage systems," in *Proc. SRDS Perform. Virtualization Large-Scale Storage Syst.*, 2003. pp. 109–118.
5. IBM Corporation, Armonk, NY, USA, IBM DB2 PureScale. [Online]. Available: http://www.ibm.com/software/data/db2/linux-unix-windows/purescale/
6. Oracle Corporation, Redwood City, CA, USA, Oracle RAC. [Online]. Available: http://www.oracle.com/technetwork/products/clustering/overview/index.html
7. IBM Corporation, Armonk, NY, USA, IBM General Parallel File System (GPFS). [Online]. Available: http://www.ibm.com/systems/software/gpfs
8. F. J. Corbato, "A paging experiment with the MultiCS system," MIT, Cambridge, MA, USA, MIT Proj. MAC Rep. MAC-M-384, May 1968.
9. IBM Corporation, Armonk, NY, USAIBM DS8000 Systems Storage. [Online]. Available: http://www.ibm.com/systems/storage/disk/ds8000/
10. IBM Corporation, Armonk, NY, USA, IBM Easy Tier. [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp4667.html
11. IBM Corporation, Armonk, NY, USA, IBM Power Systems. [Online]. Available: http://www.ibm.com/systems/power/
12. IBM Corporation, Armonk, NY, USA, IBM EXP30 Ultra SSD Drawer. [Online]. Available: http://www.ibm.com/systems/power/hardware/peripherals/ssd/details.html

13. IBM Corporation, Armonk, NY, USA, IBM FlashSystem. [Online]. Available: http://www.ibm.com/systems/storage/flash-storage/index.html
14. R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. 11th USENIX FAST*, San Jose, CA, USA, 2013, pp. 45–58.
15. S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *Proc. 28th IEEE MSST*, Pacific Grove, CA, USA, 2012, pp. 1–12.
16. Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proc. 10th USENIX FAST*, San Jose, CA, USA, 2012, pp. 1–14.
17. J. Guerra, W. Belluomini, J. Glider, K. Gupta, and H. Pucha, "Cost effective storage using extent based dynamic tiering," in *Proc. 9th USENIX FAST*, San Jose, CA, USA, 2011, pp. 273–286.
18. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network File SystemVersion 4 Protocol Specification," Internet Eng. Task Force, Fremont, CA, USA, RFC 3530, Apr. 2003.
19. T. Wong and J. Wilkes, "My cache or yours? Making storage more exclusive," in *Proc. USENIX ATC*, Monterey, CA, USA, 2002, pp. 161–175.
20. A. Arpaci-Dusseau and R. Arpaci-Dusseau, "Information and control in gray-box systems," in *Proc. 18th ACM SOSP*, Banff, AB, Canada, 2001, pp. 43–56.
21. L. Bairavasundaram, M. Sivathanu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "X-RAY: A non-invasive exclusive caching mechanism for RAIDs," in *Proc. 31st IEEE Annu. ISCA*, Washington, DC, USA, 2004, p. 176.
22. M. Mesnier, F. Chen, T. Luo, and J. Akers, "Differentiated storage services," in *Proc. 23rd ACM SOSP*, Cascais, Portugal, 2011, pp. 57–70.
23. Fusion-io ioTurbine. [Online]. Available: http://www.fusionio.com/products/ioturbine
24. Fusion-io ioCache. [Online]. Available: http://www.fusionio.com/products/iocache
25. SanDisk FlashSoft. [Online]. Available: http://www.sandisk.com/goto/flashsoft-connect
26. SanDisk ExpressCache. [Online]. Available: http://www.sandisk.com/products/software/express-cache
27. Qlogic FabricCache. [Online]. Available: http://www.qlogic.com/Pages/QLE10000/FabricCache.aspx
28. Qlogic FabricCache shared cache. [Online]. Available: http://www.qlogic.com/Resources/Documents/WhitePapers/Routers/WhitePaper_MtRainier_ValueOfSharedSSDs.pdf

*Received August 15, 2013; accepted for publication September 14, 2013*

**Sangeetha Seshadri** *IBM Research Division, Almaden Research Center, San Jose, CA 95120 USA (seshadrs@us.ibm.com).*
Dr. Seshadri is a Research Staff Member in the Storage Research Department at the IBM Almaden Research Center. She received a B.E. (honors) degree in computer science and M.S. (honors) degree in mathematics from the Birla Institute of Technology and Science (BITS), Pilani, India, in 2002. She received her Ph.D. degree in computer science from the Georgia Institute of Technology, Atlanta, in 2009. She has previously worked at Oracle Corporation and Microsoft and has been with IBM since 2009. Her areas of interest include storage architectures for high availability, distributed systems, and workload analytics.

**Paul H. Muench** *IBM Research Division, Almaden Research Center, San Jose, CA 95120 USA (pmuench@us.ibm.com).*
Mr. Muench is a Senior Technical Staff Member in the Storage Research Department at the IBM Almaden Research Center. He received a B.S. degree in computer science from Purdue University in 1987. Directly from Purdue, he joined the systems division of IBM and subsequently the IBM Research Division. Mr. Muench has made significant contributions in the areas of storage virtualization, caching, and tiering resulting in four IBM Outstanding Technical Achievement Awards and six patents.

**Lawrence Chiu** *IBM Research Division, Almaden Research Center, San Jose, CA 95120 USA (lchiu@us.ibm.com).*
Mr. Chiu is a Distinguished Engineer and manager in the Storage Research Department at the IBM Almaden Research Center. Mr. Chiu received his M.S. degree in computer engineering from the University of Southern California in 1991 and his M.S. degree in technology commercialization from the McCombs School of Business, University of Texas, Austin, in 2003. He currently manages the Scalable Storage Systems Group at the Almaden Research Center, focusing on highly scalable and highly available enterprise storage systems, as well as solid-state storage architectures.

**Ioannis Koltsidas** *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (iko@zurich.ibm.com).*
Dr. Koltsidas is a Research Staff Member in the Storage Technologies Department at the IBM Research - Zurich lab. He received a B.S. degree in electrical and computer engineering from the National Technical University of Athens, Greece, and a Ph.D. degree in computer science from the University of Edinburgh, United Kingdom. He subsequently joined IBM Research - Zurich, where he has been working on Flash-based storage, tape storage, and distributed file systems.

**Nikolas Ioannou** *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (nio@zurich.ibm.com).*
Dr. Ioannou is a Research Staff Member in the Storage Technologies Department at the IBM Research - Zurich lab. He received his Ph.D. degree in computer science from the University of Edinburgh, United Kingdom. He received a B.S. degree in electrical and computer engineering from the National Technical University of Athens, Greece. He joined IBM in 2012. His areas of interest include scale-out and highly available enterprise storage systems, as well as storage-class memories.

**Robert Haas** *IBM Corporate Headquarters, Armonk, NY 10504 USA (rha@zurich.ibm.com).*
Dr. Haas is in charge of storage strategy in the IBM Corporate Strategy Department. Prior to this assignment, he led research in storage security, distributed storage, and tape- and Flash-based systems at the Storage Technologies Department at the IBM Research - Zurich lab. He received a Ph.D. degree from the Swiss Federal Institute of Technology (ETH), Zurich, in 2003, and an M.B.A. degree from Warwick Business School, United Kingdom, in 2011, and has authored over 60 scientific publications, technical standards, and patents.

**Yang Liu** *IBM China, Shanghai, 201203 China (liuyang@cn.ibm.com).*
Mr. Liu is a Senior Software Engineer in the DS8000* development department at the IBM GCG STG (Greater China Group, Systems and Technology Group) lab. Mr. Liu received his M.S. degree in computer engineering from Xi'an JiaoTong University in 2006. He currently works on the DS8000 development, focusing on Flash and tiering system integration.

**Mei Mei** *IBM China, Shanghai, 201203 China (meimei@cn.ibm.com).* Ms. Mei is a Staff Software Engineer in the DS8000 development department at the IBM GCG STG (Greater China Group, Systems and Technology Group) lab. Ms. Mei received her M.S. degree in communication and information system from Beijing University of Posts and Telecommunications in 2010. She currently works on DS8000 development, focusing on Easy Tier.

**Stephen Blinick** *IBM Tucson, Tucson, AZ 85744 USA (blinick@us.ibm.com).* Mr. Blinick is a Senior Technical Staff Member in the Enterprise Storage Development in Tucson, Arizona. He received a B.S. degree in computer science from the University of Arizona in 1999. Mr. Blinick has worked in IBM enterprise storage development for over 12 years as a microcode developer, PCIe fabric architect, and lead architect for several DS8000 releases. He is currently a lead architect of the DS8870, focusing on future Flash technology integration and analytics.