Architectures for Controller Based CDP

Guy Laden *Paula Ta-Shma* *Eitan Yaffe* *Michael Factor*
Shachar Fienblit
IBM Haifa Research Laboratories
{*laden, paula, eitany, factor, shachar* }*@il.ibm.com*

Abstract

Continuous Data Protection (CDP) is a recent storage technology which enables reverting the state of the storage to previous points in time. We propose four alternative architectures for supporting CDP in a storage controller, and compare them analytically with respect to both write performance and space usage overheads. We describe exactly how factors such as the degree of protection granularity (continuous or at fixed intervals) and the temporal distance distribution of the given workload affect these overheads. Our model allows predicting the CDP overheads for arbitrary workloads and concluding the best architecture for a given scenario. Our analysis is verified by running a prototype CDP enabled block device on both synthetic and traced workloads and comparing the outcome with our analysis. Our work is the first to consider how performance is affected by varying the degree of protection granularity, both analytically and empirically. In addition it is the first to precisely quantify the natural connection between CDP overheads and a workload's temporal locality. We show that one of the architectures we considered is superior for workloads exhibiting high temporal locality w.r.t. granularity, whereas another of the architectures is superior for workloads exhibiting low temporal locality w.r.t. granularity. We analyze two specific workloads, an OLTP workload and a file server workload, and show which CDP architecture is superior for each workload at which granularities.

1 Introduction

Continuous Data Protection (CDP) [13] is a new paradigm in backup and recovery, where the history of writes to storage is continuously captured, thereby allowing the storage state to be potentially reverted to any previous point in time. Typically the amount of history stored is limited by the operator, either in temporal terms using a *CDP window* (e.g. 2 days) or in terms of the amount of additional storage available for history data. CDP can be provided by different entities in the I/O path such as the host being protected (by the filesystem [28, 27, 24] or the Logical Volume Manager (LVM)), a SAN appliance [7], a SAN switch or the block storage controller. We focus on CDP enabling a block storage controller although parts of our work may be applicable elsewhere. We focus on the block level since this is typically the lowest common denominator of real world heterogeneous applications. We focus on the controller setting because it allows a wider range of architectures and has potential for performance and resource usage benefits. For example reverts do not generate network I/O, and both device I/O and space can potentially be reduced.

Advanced functions such as (writable) point-in-time copies and remote replication have already been introduced to storage controllers. The advent of these and other features has resulted in a growing divergence between the notion of logical and physical volumes. Moreover, log structured file systems [25] and log structured arrays (a similar notion realized in a controller setting) [22] have proposed exploiting this notion improve file system/controller performance for write requests by turning logically random write requests to physically sequential ones. CDP is an additional technology that furthers this trend, by introducing time as an additional dimension that can be virtualized. In this paper we study some of the associated performance tradeoffs.

Enabling the controller to save the history of host writes typically requires that the writes are duplicated and split. We present four different architectures for controller-based CDP that are differentiated by the point in time at which the writes are split: on receipt of the write by the controller, on destage of the write from the controller cache, on overwrite of a block by a newer version or never (avoiding duplication/splitting).

There is an ongoing debate [1] as to whether *continuous* protection (aka every write protection) provides

much benefit to customers compared to coarser protection granularities. As a result some products going under the banner of CDP provide continuous (aka every write) protection, while other products are said to support *near* CDP, with protection granularities ranging from minutes to hours. CDP at whatever granularity does not come for free in either performance (impact on I/O throughput and latency) or disk space. We provide an exact analysis of the way the degree of protection (from granularity of every write to arbitrarily large fixed intervals) affects performance and space overheads.

When measuring performance, we focus on the count of additional device I/O's incurred by write requests in regular (non revert) scenarios for keeping track of prior versions and recording them in the CDP history. We focus on write requests since this is the main CDP overhead in good-path scenarios for 3 out of 4 of our architectures, where read performance is equivalent to that of a regular volume. Performance immediately after a revert depends on the data structures chosen to represent the CDP history and is the same across the architectures we compare since they share a common infrastructure for storing CDP history. Note that for appliance based block CDP, duplicating and splitting writes cannot be done at destage or overwrite time. Therefore our comparison of architectures also sheds light on inherent potential differences between appliance and controller based CDP.

Apart from logging write traffic, a controller offering a high-end CDP feature set must provide for fast revert of a production LUN to a prior point in time, be able to quickly export writable historical versions of the production volume, to perform automatic space reclamation of the older historical data etc. Although our proposed architectures all support these features, in this paper we do not describe the details and focus on the overheads CDP incurs when providing basic block I/O functionality.

We analyze our CDP architectures to accurately predict both the count of additional I/O's incurred by write requests and the space consumption overhead of each architecture. We describe exactly how factors such as the degree of protection granularity (continuous or at fixed intervals) and the temporal distance distribution of the given workload affect these overheads. Our work precisely quantifies the natural connection between CDP overheads and a workload's temporal locality. It allows predicting the CDP overheads for arbitrary workloads and concluding the best architecture for a given scenario.

We implemented a prototype CDP enabled block device and use it to validate our analysis against implementations of the architectures. We compare the cost of the architectures on real-world filesystem traces and synthetic OLTP traces, as we vary the protection granularity, and we conclude the best CDP architectures for these scenarios.

The outline of the paper is as follows. In section 2 we describe architectures for implementing CDP in a storage controller and in section 3 we characterize their performance analytically. Section 4 presents an evaluation of the performance of the proposed architectures on synthetic and real-life traced workloads. Section 5 reviews related work, and in section 6 we present our conclusions.

2 Architectures

2.1 Controller Background

Modern storage controllers typically contain a combination of processor complexes, read and fast write cache, host and device adapters, and physical storage devices. All components are typically paired so that there is no single point of failure [18, 17, 16]. Our figures describe a configuration having two *nodes*, where each one has one processor complex, one or more host adapters, one or more device adapters and one read and fast write cache, and each node owns a set of volumes (logical units (LUNs)) of the physical storage. To implement the fast write cache a subset of the total cache memory is backed by non volatile storage (NVS) on the opposite node. On node failure, volume ownership is transferred to the opposite node.

Cache (whether backed by NVS or not) is typically divided into units called *pages*, whereas disk is divided into units called *blocks* - pages/blocks are the smallest units of memory/disk allocation respectively. Stage and destage operations typically operate on an *extent*, which is a set of consecutive blocks. In the context of this paper, we concentrate on the case where the extent size is fixed and is equal to the page size. All architectures we discuss can be generalized to deal with the case that the extent size is some multiple of the page size although this requires delving into many details which need to be addressed for all architectures, and this is orthogonal to the main ideas we want to develop in the paper.

2.2 Integrating CDP into a Controller

We assume some mechanism for mapping between logical and physical addresses [1] where there is not a simple a priori relationship between them. Thin provisioning is another controller feature which can benefit from such a mapping. Such a mechanism must allow the cache to stage/destage from a physical address which differs from the logical address of the request. One way to do this is to allow the cache to invoke *callbacks*, so that a logical to physical map (LPMap) can be accessed either before

or after stage and destage operations. In the case of CDP, timestamps will play a role in the LPMap.

We should distinguish between volumes that are *directly addressible*, and those which are *mapped* since they require a structure such as an LPMap in order to access them. Mapped volumes may be stored in individual physical volumes or alternatively several of them can be stored together in a larger physical volume which serves as a storage pool. Mapped volumes may sometimes be *hidden* from the user, as we will see for some of the architectures.

The LPMap structure supports the following API:

insert insert a mapping from a particular logical address to physical address, to be labeled by the current timestamp

lookup look up the current physical address corresponding to a particular logical address

revert revert the LPMap structure to a previous point in time

All architectures we consider will use the same LPMap representation.

2.3 Architectural Design Points

There are several factors which determine the CDP architecture. One factor is whether the current version of a volume is stored together with the historical data or separately from it. Storing current and historical data together results in the *logging* architecture. Note that this is not an option with host and network based CDP, where writes are duplicated and split at the host or at the network.

Since good sequential read performance is likely to be essential for most workloads, we consider alternative architectures containing 2 volumes - a directly addressible volume to hold the current version, and a hidden, mapped volume to contain historical data. In order to restore good sequential read performance, reverting such a volume to a previous point in time now inherently requires significant I/O - each changed extent needs to be physically copied from the history store to the current store. This I/O activity can be done in the background while the history store is used to respond to read requests as needed. This is similar to a background copy feature which accompanies some point-in-time copy implementations [10], and we expect the duration of such a background copy and the degree of performance degradation to be similar.

Assuming this separation between current and historical data, an important factor is when the duplication and splitting of I/O's is done. Splitting at write time leads to the *SplitStream* architecture, splitting at destage time leads to the *SplitDownStream* architecture, and splitting before overwrite time leads to the *Checkpointing* architecture. The corresponding location of the splitting would be above the cache, below the cache and at the storage respectively.

Note that our figures depict the current and history volumes as owned by opposite nodes, even though this is not necessarily the case. Moreover, in our figures we depict the CDP architectures as implemented on the previously described controller hardware. The figures might look slightly different if we were to design each particular CDP architecture from scratch with its own dedicated hardware.

A need may arise for *multi-version cache* support, namely the cache may need the ability to hold many versions of the same page. The CDP granularity requested by the user determines when data may be overwritten, for example every write granularity means that all versions must be retained and none overwritten. In this case, if the cache cannot hold multiple versions of a page, then an incoming write can force a destage of an existing modified cache entry. This needs to be taken into account since it can have affect the latency of write requests. How to best implement a multi-version cache is outside the scope of this paper.

2.4 The Logging Architecture

The logging architecture is the simplest - the entire history of writes to a volume is stored in a mapped volume which is not hidden from the user. Figure 1 depicts the

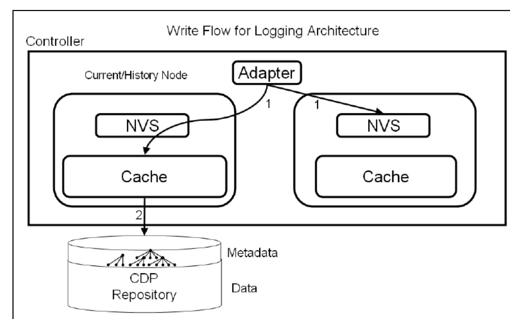

Figure 1: Logging Architecture Write Flow - each write request incurs at most 1 user data device I/O.

write flow for the logging architecture. Note that each write request incurs at most 1 user data device I/O. If the CDP granularity is coarser than the rate of writes to a particular logical address, then there is potential to avoid this I/O if the logical address is cached.

Both stages and destages need to access the LPMap structure. This approach is good for a write dominated

workload, since not only do we avoid additional device I/O as a result of duplicating or copying from current volumes to historical volumes, we also have an opportunity to covert random writes into sequential ones, as done in [25]. An additional benefit is that with the right data structures for representing CDP history, revert can be done with a small constant overhead although the details are out of scope for this paper. Note that both reads and writes may require accessing the LPMap structures, although this does not always require additional device I/O, if these structures are intelligently cached. More importantly, the cost of accessing the LPMap is similar for all the architectures we discuss.

The important downside of this architecture is that one is likely to forfeit good sequential read performance, mainly because of the difficulty of sequential layout of extents on disk, and also because of access to the LPMap meta data. Although the layout is dependent on the implementation of the LPMap, which is out of the scope of this paper, guaranteeing good sequential read performance for this type of architecture is a difficult research problem [25]. Our simpler approach for achieving good sequential read performance is to separate current and historical data. Sequential read performance of historical data is not critical since it is only read immediately after a revert.

An additional issue with the logging architecture is that it is not straight forward to CDP enable an existing volume. Building the entire LPMap structure up front is not feasible, although if we choose to build it on demand, we may wait indefinitely until we can recycle the space of the original volume.

Note that the logging architecture is appropriate for a logging workload, since read access is only needed in error scenarios.

2.5 The SplitStream Architecture

Duplicating and splitting data above the cache leads to the *SplitStream* architecture. One copy of the data is sent to a *current store* - a directly addressable volume which holds the current version, and an additional copy is sent to a *history store* - a hidden, mapped volume, which in this case contains both current and previous versions of the data.

Figure 2 depicts the write flow for the SplitStream architecture. Note that each write request incurs at most 2 user data I/Os, one to each of the current and history stores. Just as for regular volumes, the cache can potentially save a current store I/O for those logical addresses that are repeatedly written to while they are still cache resident, and this effect is independent from the CDP granularity. If the CDP granularity is coarser than the rate of writes to a particular logical address, then there is potential to save a second I/O if that logical address is cached at the history store. Compared to the logging architecture, we gain good sequential read performance at the expense of incurring additional I/Os on write requests, and using more resources such as disk space and cache memory.

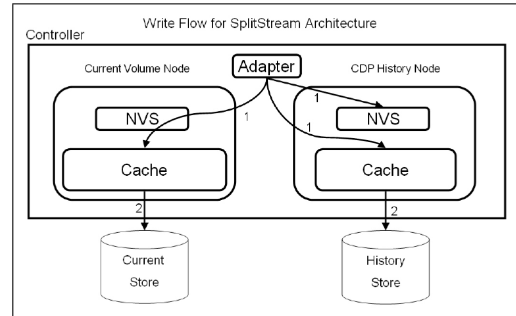

Figure 2: SplitStream Architecture Write Flow - Writes are split above the cache, and each write request incurs up to 2 user data device I/O's.

Note that in this case the cache can manage each volume separately, so a multi-versioned cache may not be a necessity. The current store does not need to deal with historical data, so can be cached as a regular volume, while the history store will not service reads, so caching simply serves as a means to buffer write access. Note that for coarse granularity the history store cache also serves to reduce device I/O for those logical addresses that are written to frequently. In the SplitStream architecture, the same data may appear twice in cache which inflates memory requirements, although the relative sizes of the current and history caches and the cache replacement algorithm could be tailored to take this into account.

2.6 The SplitDownStream Architecture

In the SplitDownStream architecture, the duplicating and splitting of data occurs under the cache, at destage time, instead of above the cache as is the case for SplitStream. In other respects the architectures are identical. This allows cache pages to be shared across current and historical volumes, thereby conserving memory resources. However, since the cache functions both to serve read requests and to buffer access to the history store, a versioned cache is needed to avoid latency issues.

Figure 3 depicts the write flow for the SplitDownStream architecture. Note that if we ignore cache effects which are different in the two architectures, SplitDownStream is identical to SplitStream in the number of device I/O's incurred by a write request.

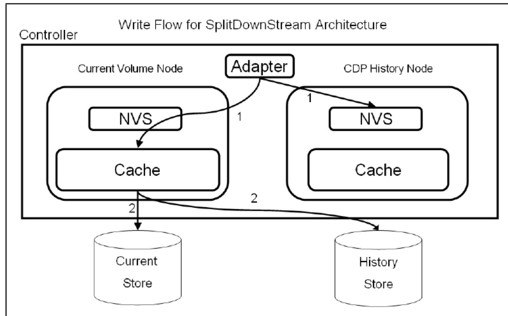

Figure 3: The SplitDownStream Architecture Write Flow - writes are split below the cache and each write request incurs up to 2 user data device I/O's.

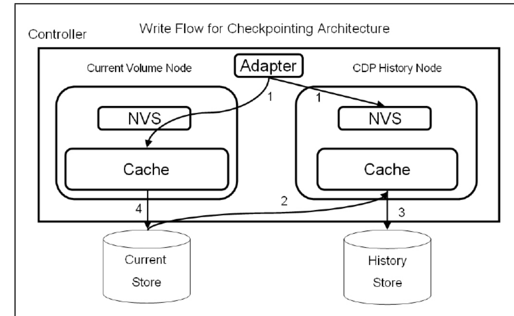

Figure 4: The Checkpointing Architecture Write Flow - the previous version is copied to the history store before being overwritten. Each write request incurs up to 3 user data device I/O's.

2.7 The Checkpointing Architecture

Similar to the SplitStream and SplitDownStream architectures, the Checkpointing architecture has a directly addressible current store and a hidden mapped history store. However, in this case the current version exists only at the current store, while the history store contains only previous versions of the data. At destage time, before overwriting an extent at the current store, we first check whether it needs to be retained in the history store. This depends on the CDP granularity - every write granularity mandates that all versions need to be retained, whereas a coarse granularity requires very few versions to be retained. If needed, the extent is copied to the history store before being overwritten by the destage. Note that copying an extent from the current store to the history store requires 2 device I/O's - one to stage to the history store cache, and another to destage to the history store. Thus in total we may incur up to 3 device I/O's per write request. Figure 4 depicts the write flow for the Checkpointing architecture. If we do not use NVS at the history store, all 3 device I/Os are synchronous to the destage operation, although not to the write request itself, since we cannot destage until the previous version of the extent is safely on disk at the history store. Therefore to provide good latency a versioning cache is essential.

Of the total 3 possible device I/O's per write request, 2 of these (those for copying from the current to the history store) are not at all influenced by caching, and are completely determined by the CDP granularity. The other I/O (for destaging to the current store) can be avoided by caching if the particular version can be discarded according to the CDP granularity.

We point out that the checkpointing architecture is essentially an extension of the popular copy on destage technique used to implement Point In Time (PIT) copies [12] to the CDP context.

3 Analysis

In this section we analyze both the number of device I/O's incurred by write requests and the space overheads of the various CDP architectures, as a function of the CDP granularity.

3.1 Preliminaries

The time dimension can be divided into a set of fixed length **granularity windows** of size g. The requirement is to retain the last write in each granularity window. The smaller the value of g, the finer the granularity. When g is arbitrarily small, this results in every write CDP, whereas when g is arbitrarily large, this results in a regular volume with no CDP protection.

We define a write to a particular logical address to be **retained** if it is the last in its granularity window at that address. Fine granularity gives a large proportion of retained writes, while the opposite is true for coarse granularity. We use r to denote the proportion retained writes ($0 \leq r \leq 1$). For example, every write granularity gives $r = 1$ and as we increase g so that the granularity becomes coarser, r approaches 0. [2]

With respect to a fixed granularity, increasing the temporal distances between writes decreases the probability of a write to be retained. For a trace of a given workload, the **temporal distance distribution** of the workload is a function T, where for a given temporal distance t, $T(t)$ is the fraction of writes [3] with distance $\leq t$ to the subsequent write, so $0 \leq T(t) \leq 1$. There are a number of examples of the use of temporal distance distribution graphs in the literature [26, 32].

We use c to take write caching into account, where c is the proportion of writes which are *evicted* from cache before being overwritten ($0 \leq c \leq 1$). $c = 1$ means no absorption of writes by cache, whereas $c = 0$ means

all writes can potentially be absorbed. Note that $1-c$ is the write cache hit rate. Like the cache hit rate, c depends both on attributes of the write cache namely its size and replacement algorithm, as well as on attributes of the workload, namely its temporal locality and the distribution of its writes across logical addresses. The distribution of writes has a clear effect since repeated writes to a single address require less cache space than spreading the same number of writes across many addresses [4].

We define a write to be **incurred** if it is either retained, or evicted from cache before being overwritten, or both. In the logging architecture, the incurred writes are those which lead to a device I/O. We use d to denote the proportion of incurred writes under the logging architecture ($0 \leq d \leq 1$). It is important to note that in general $d \neq r + c - rc$ since being evicted from cache and being retained are not necessarily independent events.

3.2 Device I/O's Incurred by Write Requests

For a given r, c and d, figure 5 summarizes the I/O's incurred for the various architectures per write request. Note that both r and d are dependent on the granularity g. In section 3.4 we show how to derive r, and in section 3.5 we show how to derive d.

For example, if we have a workload with flat temporal locality of 0.5 seconds, then a once per second granularity gives $r = 0.5$. If we assume no cache then the expected cost of checkpointing per write request is $rp + (1 - r)p'$, where p is the cost for a retained write and p' is the cost for a non retained write in the checkpointing architecture. This equals $3r + (1-r) = 2r + 1$. However for SplitStream and SplitDownStream the cost is always 2. The turning point is at $r = 0.5$, a larger value of r will give an advantage to Split(Down)Stream, whereas a smaller value will by advantageous to Checkpointing. r is determined by the relationship between the temporal locality of a given workload and the CDP granularity chosen by the user. Note that r is oblivious to the distribution of the writes, since the same fraction of retained writes can be as easily obtained with the same number of writes to a single logical address as to 1000 addresses, only the relationship of the temporal distance of the writes and the granularity chosen is important.

Caching does have an effect, and it should be clear from figure 5 that Split(Down)Stream can take better advantage of the cache than Checkpointing. Checkpointing can utilize cache to save non retained I/O's at the current store, but Split(Down)Stream can do the same and more. At an unrealistic extreme of infinite cache, SplitStream reduces I/O's to r per write request, the bare minimum (and equal to the Logging architecture). Here the expected cost of checkpointing per write request is $3r$, since all retained writes reach the history store via the current store. Note that SplitDownStream is slightly different from SplitStream in its behavior on retained I/O's at the current store. Since we destage cache entries to the current and history stores together, all retained writes reach the current store. In summary, increasing the size of the cache reduces c, and the SplitStream architecture gains most benefit from this, closely followed by the SplitDownStream architecture.

Unlike the case for regular volumes ($r = 0$), as r increases, the benefit which can be obtained by increasing the cache size becomes more limited. For example, for the extreme case of $r = 1$ e.g. every write granularity, the I/O cost for Checkpointing per write is 3 independent of the cache size, while for SplitDownStream it is 2. There is no crossover in this case and SplitDownStream always dominates. At the opposite extreme of a regular (non CDP enabled) volume ($r = 0$), the cost of checkpointing is c, whereas for Split(Down)Stream it is $2c$, giving a crossover at $c = 0$. This means that Checkpointing dominates Split(Down)Stream for all values $c > 0$ e.g. for an arbitrarily large cache. The reason is that Split(Down)Stream always splits and duplicate all writes, so it costs more than a regular volume when cache entries are destaged. Checkpointing, however, behaves like a regular volume in this case.

In general the crossover between Checkpointing and SplitStream is obtained at $c = 2r$, where if $c > 2r$ then Checkpointing dominates. This means that the proportion of writes evicted from cache before being overwritten needs to be at least twice the proportion of retained writes in order for Checkpointing to dominate. Since this is impossible when $r > 0.5$, Split(Down)Stream always dominates in that scenario.

3.3 Space Overhead

The fraction of retained writes r also determines the space needed to hold the CDP history, so there is a natural relationship between performance in terms of I/O counts incurred by writes and CDP space overhead. Figure 6 summarizes the space overhead for the various architectures. w denotes the number of writes within a given CDP window, a denotes the size of the addressable storage, and f denotes the size of the storage actually addressed during the CDP window. Since the Logging architecture is space efficient, it can have a lower space cost than a regular volume, whereas the cost of Split(Down)Stream equals the cost of a regular volume and a Logging volume combined. Checkpointing saves some space overhead since the current version is not stored in the history store.

		Logging	Checkpointing	SStream	SDownStream	Crossover
retained I/O's	current store		1	c	1	
	history store		2	1	1	
	sub total $(r = 1)$	1	3	$1 + c$	2	never
evicted I/O's	current store		c	c	c	
	history store		0	c	c	
	sub total $(r = 0)$	c	c	$2c$	$2c$	$c = 0$
total I/O's	**no cache** $(c = 1)$	1	$2r + 1$	2	2	$r = 0.5$
	infinite cache $(c = 0)$	r	$3r$	r	$2r$	$r = 0$
	arbitrary cache	d	$d + 2r$	$d + c$	$2d$	$c = 2r$

Figure 5: A table of device I/O's incurred per write request in terms of r, the proportion of retained writes and c, the proportion of writes which are destaged before being overwritten, and d, the proportion of writes resulting in a device I/O under the logging architecure. We show the analytical crossover point between Checkpointing and SplitStream Architectures.

Architecture	Space Overhead
Regular	a
Logging	rw
Split(Down)Stream	$a + rw$
Checkpointing	$a + rw - f$

Figure 6: Space overhead of CDP architectures in terms of r, w, a and f, where w is the total number of writes, r is the proportion of retained writes, a is the size of the addressable storage and f is the size of the storage actually addressed.

3.4 Retained Writes as a Function of the Granularity

We already mentioned that the proportion of retained writes r depends on the relationship between the temporal distance between writes and the granularity. Given a temporal distance distribution T, we show how to express r in terms of T and the granularity g. This allows us to infer properties of both device I/O counts incurred by writes as well as the space overhead of CDP for any given workload according to its temporal distance distribution.

We divide the time dimension into a set of fixed length granularity windows of size g. We assume that the first window starts at a uniform position between 0 and g. Let $R(g) = E(r(g))$ where E denotes expectation.

Claim 3.1

$$R(g) = 1 - \frac{1}{g} \int_{t=0}^{g} T(t)dt$$

Proof 3.1 *Let $w_1, \ldots, w_n$ be the set of writes in our given trace, and let a_i be the temporal distance between w_i and the subsequent write at the same logical address. Let X_i be the random variable that is 1 iff w_i is last in its granularity window, and so is a retained write. We have that $R(g) = E(\frac{1}{n}\sum_{i=1}^{n} X_i) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$.*

Now, $X_i = 1$ and w_i is a retained write iff it is within distance a_i from the end of the window. Since we place the start of the first granularity window at a uniform offset, we have that $E(X_i) = min(\frac{a_i}{g}, 1)$. Thus, $E(X_i) = \frac{1}{g}\int_{t=0}^{g} f_i(t)$, where $f_i(t) = 1$ if $t \leq a_i$ and is 0 otherwise.

Altogether, $R(g) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{g}\int_{t=0}^{g} \frac{1}{n}\sum_{i=1}^{n} f_i(t)$ and $\frac{1}{n}\sum_{i=1}^{n} f_i(t) = 1 - T(t)$.

3.5 I/O Counts as a Function of the Granularity

Given a trace's temporal distance distribution T, we show how to express d as a function of g, assuming a certain write cache hit rate $(1 - c)$ and under a certain model of cache replacement. In order to simplify our analysis we will assume a Least Recently Written (LRW) replacement policy for the write cache, and will assume that the I/O rate is uniform. This means that writes wait in cache in a FIFO queue to be destaged, and it follows that the time interval writes wait from their last access until they are evicted is roughly constant for all writes. Given a cache hit rate $(1 - c)$, we define the *cache window* s to be the time interval that satisfies $c = 1 - T(s)$ and therefore is a result of this hit rate under our model.

Recall that d is the proportion of incurred device I/O's in the logging architecture. A write can result in a device I/O either because it is retained, or it is evicted from cache before being overwritten, or both. Recall that in general $d \neq r + c - rc$ since being evicted from cache and being retained are not necessarily independent events.

We show how to express d as a function of the granularity g and the cache window size s, where $c = 1 - T(s)$:

Claim 3.2 *Let R be as defined in claim 3.1.*

$$D(g,s) = \begin{cases} \frac{s}{g}R(s) + c(1-\frac{s}{g}) & \textit{if } s < g \\ R(g) & \textit{otherwise} \end{cases}$$

Proof 3.2 *Let $w_1, \ldots, w_n$ be the set of writes in our given trace, and let a_i be the temporal distance between w_i and the subsequent write at the same logical address. Let X_i be the random variable that is 1 iff w_i is an incurred write. A write w_i is incurred if it is either last in its granularity window or $a_i > s$ which causes the write to be evicted from the cache. If $s \geq g$, then $X_i = 1$ iff w_i is a retained write, because $a_i > s$ implies $a_i > g$ so all evicted writes are also retained. Therefore in this case $D(s,g) = R(g)$.*

Assuming $s < g$, we have $D(g,s) = E(\frac{1}{n}\sum_{i=1}^{n} X_i) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$. Since we place the start of the first granularity window at a uniform offset, we have:

$$E(X_i) = \begin{cases} \min(\frac{a_i}{g}, 1) & \textit{if } s > a_i \\ 1 & \textit{otherwise} \end{cases}$$

We define a function $f_i(t) = 1$ if $t \leq a_i$ and is 0 otherwise, and a function $\bar{f}_i(t) = f_i(t)$ if $t \leq s$ and $\bar{f}_i(t) = f_i(s)$ otherwise. Thus, we get $E(X_i) = \frac{1}{g}\int_{t=0}^{g} \bar{f}_i(t)$.

Altogether we get: $D(g,s) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{g}\int_{t=0}^{g} \frac{1}{n}\sum_{i=1}^{n} \bar{f}_i(t) = \frac{1}{g}[\int_{t=0}^{s} \frac{1}{n}\sum_{i=1}^{n} f_i(t) + \int_{t=s}^{g} \frac{1}{n}\sum_{i=1}^{n} f_i(s)]$. Since we know that for any $t \in [0,g]$ it holds that $\frac{1}{n}\sum_{i=1}^{n} f_i(t) = 1 - T(t)$ we get $D(g,s) = \frac{1}{g}[\int_{t=0}^{s}(1-T(t)) + \int_{t=s}^{g}(1-T(s))] = \frac{1}{g}[sR(s) + (g-s)(1-T(s))]$.

This result can be used to calculate the I/O counts for the other architectures. Note that once the $s = g$, further increasing the cache size will not affect d and so will not provide additional benefit. This means that for fine granularity, cache size does not play an important role.

3.6 Discussion

We chose the LRW caching policy because it is simple to analyze and to implement. The result is a baseline comparison between the various architectures. Other caching policies such as WOW [15] could be considered and possibly adapted to the CDP context - this is a topic for further work. In practice the cache management policies implemented by modern storage controllers are complex and involve techniques such as prefetching and write coalescing. We did not introduce these aspects to our model however in order to keep it simple.

Our analysis applies to a segregated fast write cache, and an important topic of further work is to generalize it to the non segregated case which models the implementation of some storage controllers.

We already mentioned that a versioned cache is needed to avoid latency issues for all architectures except for SplitStream. Assuming a versioned cache, then some of the I/O's incurred by a write request are synchronous to the *destage* operation, but not to the write operation, unless the destage is synchronous to the write (such as when the write cache is full). Even though I/O's which are synchronous to a destage may not affect latency, they limit the freedom of the controller and may effect throughput. These are topics for further work.

Since there is an order of magnitude difference between random and sequential I/O bandwidth, the degree of sequentiality of the incurred I/O's also needs to be taken into account. For both sequential and random reads, Split(Down)Stream and Checkpointing architectures behave similarly. Regarding random writes, both architectures have the potential to convert random I/O to sequential I/O at the history store. However, for sequential write workloads Checkpointing seems to have a disadvantage, since on destage previous versions may need to be copied to the history store synchronously to the destage, which interferes with the sequential I/O flow to disk. This can be somewhat offset by optimizations such as proactively copying many adjacent logical addresses to the history store together. A more detailed analysis of effect of the sequentiality of workloads on the performance of the various architectures, and an empirical evaluation, is outside the scope of this paper and is a topic for further work.

It would be interesting to do a bottleneck analysis of the various architectures, although this is beyond the scope of our work. One point to consider is the timing of the incurred device I/O's and the resulting effect on the back-end interconnect. In the Checkpointing architecture, we may see bursts of activity once a CDP granularity window completes, whereas in the other architectures the additional load is more evenly spread over time.

4 Performance

In this section we analyze our CDP architectures in the context of both synthetic and real life workloads. We analyze the properties of the workloads that affect both CDP performance and space usage, as well as empirically measuring the performance of the various architectures using a prototype CDP enabled block device.

4.1 Experimental Setup

To evaluate the CDP architectures we presented in section 2, we implemented a prototype stand alone net-

work storage server with CDP support. The prototype was written in C under Linux and offers a block storage I/O interface via the NBD protocol [5], and can also be run against trace files containing timestamped I/O's. The prototype has a HTTP management interface which allows reverting the storage to previous points in time. The CDP history structures were implemented using the B-Tree support in the Berkeley DB database library [6]. Our prototype has been tested extensively using a python test suite and has also been used to mount file systems.

Our prototype emulates a storage controller's cache with a LRU replacement policy for reads and LRW (least recently written) policy for writes. In a typical storage controller, dirty data pages are battery backed. In the controllers we model, for cost reasons, the number of these pages is limited to a small fraction of the total pages in the cache [17]. In our experiments we limited dirty pages to occupy at most 3% of the total cache size, and we varied the total cache size in our experiments. Based upon our experience, a ratio close to 3% is often seen in systems which have segregated fast write caches. We chose a page size and extent size of 4KB, and varied the granularity from every write granularity, across a range of granularity values.

4.2 Workloads

4.2.1 The SPC-1 Benchmark

Storage Performance Council's SPC-1 [9] is a synthetic storage-subsystem performance benchmark. It works by subjecting the storage subsystem to an I/O workload designed to mimic realistic workloads found in typical business critical application such as OLTP systems and mail server applications. SPC-1 has gained some industry acceptance and storage vendors such as Sun, IBM, HP, Dell, LSI-Logic, Fujitsu, StorageTek and 3PARData among others have submitted results for their storage controllers [8]. The benchmark has been shown to provide a realistic pattern of I/O work [20] and has recently been used by the research community [21, 15].

We compared the CDP architectures on workloads similar to ones generated by SPC-1. We used an earlier prototype implementation of the SPC-1 benchmark that we refer to as SPC-1 Like. The choice of a synthetic workload enabled us to monitor the effect of modifying workload parameters which is important for reaching an understanding of the behavior of the CDP architectures. The SPC-1 Like prototype was modified to generate a timestamped trace file instead of actually submitting the I/O requests. All trace files generated were 1 hour long.

A central concept in SPC-1 is the Business Scaling Unit (BSU). BSUs are the benchmark representation of the user population's I/O activity. Each BSU represents the aggregate I/O load created by a specified number of users who collectively generate up to 50 I/O's per second. SPC-1 can be scaled by increasing or decreasing the number of BSUs.

SPC-1 divides the backend storage capacity into so-called Application Storage Units (ASUs). Three ASUs are defined: ASU-1 representing a "Data Store", ASU-2 representing a "User Store" and ASU-3 representing a "Log/Sequential Write". Storage is divided between the ASUs as follows: 45% is assigned to ASU-1, 45% to ASU-2 and the remaining 10% is assigned to ASU-3. The generated workload is divided between the ASUs as follows: 59.6% of the generated I/Os are to ASU-1, 12.3% are to ASU-2 and 28.1% to ASU-3. Finally, another attribute of the SPC-1 workload is that all I/O's are 4KB aligned.

4.2.2 cello99 traces

cello99 is a well known block level disk I/O trace taken from the cello server over a one year period in 1999. cello is the workgroup file server for the storage systems group at HP labs and the workload is typical of a research group, including software development, trace analysis and simulation. At that time, cello was a K570 class machine (4 cpus) running HP-UX 10.20, with about 2GB of main memory. We used a trace of the first hour of 3/3/1999.

4.3 Workload Analysis

4.3.1 Temporal Distance Distribution

According to our analysis in sections 3, the temporal distance distribution is a crucial property of a workload which influences both performance of the various CDP architectures in terms of I/O counts and the predicted space usage. We observed that this distribution for SPC-1 Like traces is determined by the ratio between the number of BSUs (the load level) and the size of the target ASU storage. We obtained the same distribution when the number of BSUs and ASUs is varied according to the same ratio. We obtained a set of SPC-1 Like traces with different distributions of overwrite delays by varying this ratio. Increasing the number of BSUs while keeping the storage size constant means that more activity takes place in a given unit of time, and this decreases the expected overwrite delays the workload. We chose to vary the number of BSUs rather than the target storage size since this allows us to easily keep a fixed ratio between cache and storage size. For all trace files the total capacity of the ASUs is kept constant at 100GB. To modify temporal locality we varied the number of BSUs: 33, 50, 75 and

100. Our BSU/ASU ratio of 33 BSU/100 GB is comparable with certain SPC1 vendor submissions for high end storage controllers [8], therefore we expect the temporal distance distribution to be similar.

In all our measurements we ignored I/O's to ASU-3 since it represents a purely sequential-write workload (a log) and we wanted to avoid skewing our results according to this. Such a workload is characterized by very large overwrite delays, and a logging or special purpose architecture would be most suitable for providing CDP functionality. Ideally, one could choose a CDP architecture per protected volume.

Figure 7 shows the temporal distance distribution in the SPC-1 Like and cello99 traces. Note that for the SPC-1 Like traces, as the number of BSUs is increased, the average temporal distance decreases, which indicates that for a given granularity there is more potential for I/O and space savings. Note that the temporal distances exhibited by the cello99 are much shorter than those for the SPC-1 Like traces, and this leads to an expected difference in behavior of the CDP architectures.

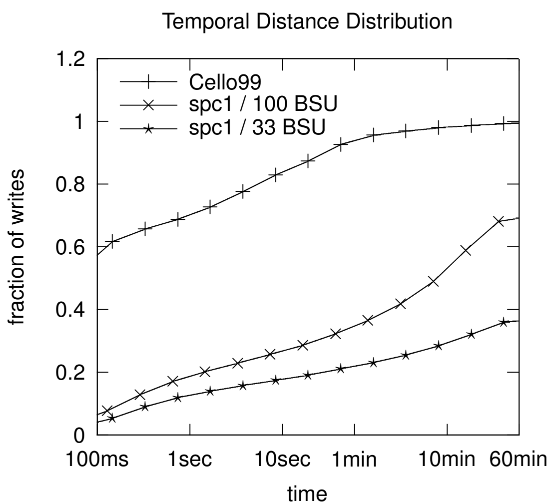

Figure 7: Temporal Distance Distribution of cello99 and SPC-1 Like workloads. For a given interval d the graph plots the fraction of writes with temporal distance $\leq d$ to the subsequent write.

4.3.2 Space Overhead and Retained Writes

Figure 8 shows the space overhead of the CDP Logging architecture as a function of granularity. The storage overhead of every write (EW) granularity is normalized to 1, and increasing the granularity reduces the space overhead. [5] Because of the relatively small temporal distances in the cello99 trace, considerable savings are possible at very fine granularities in the order of seconds. On the other hand, the relatively large temporal distances in the SPC-1 traces means that space savings are obtained only with granularities which are larger by several orders of magnitude. As shown in figure 6, the space overhead of the logging architecture equals rw, so our normalized graph is a graph of r, the proportion of retained writes, as a function of granularity. The relationship of the space overhead of the other architectures with granularity is similar, as can be derived from figure 6. We also calculated the expected proportion of retained writes analytically according to the formula from claim 3.1 and the given temporal distance distributions for the cello and SPC1 Like traces. Figure 8 compares our analytical results with our empirical ones, and we see an extremely close match, validating the correctness of our analysis.

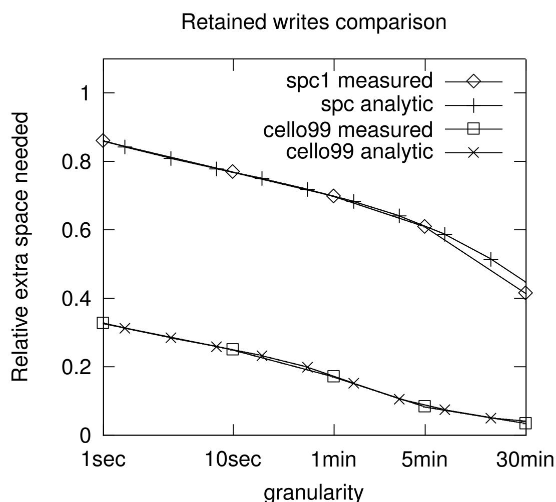

Figure 8: Tight match between analytical and empirical measurements of space overhead. The empirical measurements are normalized by dividing by the storage overhead associated with every write granularity. The analytical calculation is according to the formula in claim 3.1.

4.3.3 I/O Counts Incurred by Write Requests

We ran our prototype implementations on the Logging, Checkpointing and SplitDownStream architectures using the SPC1 Like trace with a 100BSU/100GB ratio and 4GB cache, and counted the number of I/O's incurred by write requests (see figure 9). All I/O counts are normalized according to the total number of extents written in the trace. At every write (EW) granularity, the architectures are close to a 1:2:3 ratio as expected according to our analysis. At a granularity coarser than 5 mins the Logging Architecture is close to the I/O counts of a regular volume. As the granularity becomes coarser, there is

a very gradual improvement (note the logarithmic scale of the granularity axis), and SplitDownStream dominates Checkpointing for granularities up to 30 mins, at which point there is a crossover. The gradual improvement is a result of a low proportion of writes with temporal distances up to 5 minutes, as shown in figure 7. In figure 10, we perform the same experiment on our SPC1 Like trace with a 33BSU/100GB ratio and 4GB cache. Because of the reduced temporal locality, SplitDownStream dominates Checkpointing for 60 minute granularities and beyond.

In figure 11, we plot the empirical results we obtained for the 100BSU trace against the calculated analytic results for the same trace. As can be seen there is a close match, validating our analysis.

In figure 12, we performed a similar experiment using the cello99 trace with a cache size of 64MB. As for the SPC1 Like trace, at every write (EW) granularity the architectures are close to a 1:2:3 ratio, although the higher temporal locality of this trace results in a very fast decline of I/O counts for the checkpointing architecture with an increase in granularity, and it dominates SplitDownStream at 5 minute and coarser granularities. At a granularity of 60 mins, I/O counts for checkpointing approach those Logging and of a regular volume.

In figure 13, we plot the empirical results we obtained for the cello99 trace against the calculated analytic results for the same trace. As can be seen there is a close match, validating our analysis.

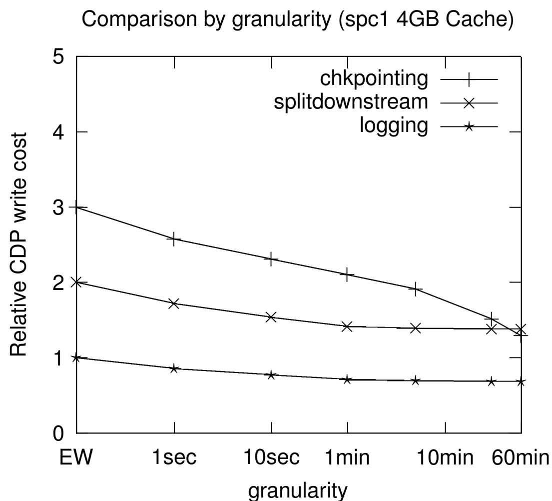

Figure 9: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for the SPC1 Like trace with 100BSU/100GB. There is a crossover point after 30mins where Checkpointing overtakes SplitDownStream - note the logarithmic scale of the X axis.

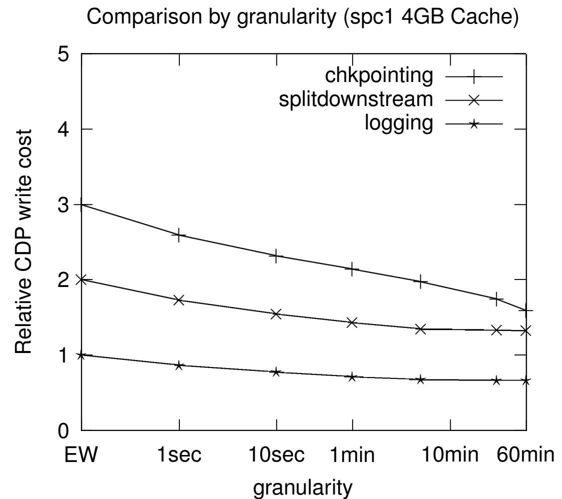

Figure 10: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for the SPC1 Like trace with 33BSU/100GB. Since this trace exhibits less temporal locality, the crossover point of figure 9 has moved to beyond 1 hour, in favor of SplitDownStream.

4.3.4 The Effect of Write Cache on I/O Counts

It is clear that increasing the write cache size reduces the I/O counts for any architecture, assuming the CDP granularity allows this. According to our analysis, the Checkpointing architecture is more limited than SplitDownStream in its ability to utilize write cache to reduce I/O counts. In figure 14, we compare the I/O counts of the various architectures as the cache size is increased, for the cello99 trace with 1 second granularity. The performance of all 3 architectures enjoy an increase in cache size, with the SplitDownStream architecture best able to utilize the additional cache space. In general we expect an increase in cache size to push the crossover point between SplitDownStream and Checkpointing to coarser granularities in favor of SplitDownStream - whereas the crossover point for 64MB in figure 12 is less than 10 minutes, for approx. 100MB it is exactly 10 minutes, and for 512MB it is greater that 10 minutes. Note that in this case a large increase in cache size makes only a small difference to the crossover granularity. Also note that once the cache size reaches approximately 256MB, a further increase in cache size does not further reduce the CDP write cost. For the 256MB cache size, we found s to be close to 10 minutes, which confirms our analysis from section 3.5 that predicted this to happen once $s = g$.

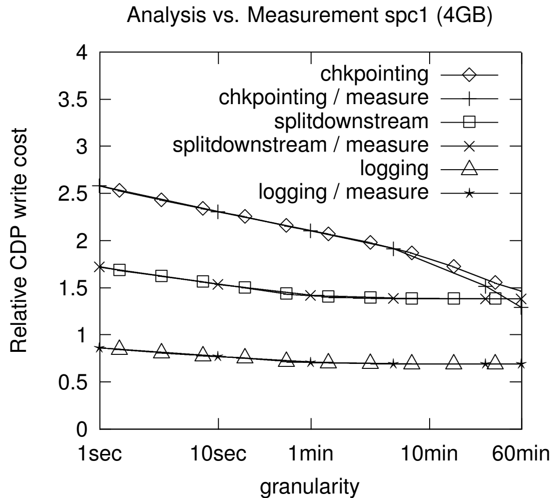

Figure 11: Tight match between analytical and empirical measurements of I/O counts for the SPC1 Like 100BSU/100GB trace. The analytical calculation is according to the analysis developed in 3.

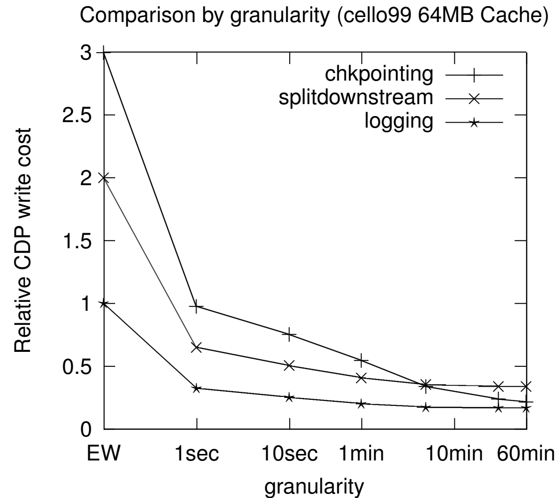

Figure 12: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for cello99 with 64MB cache. Checkpointing dominates SplitDownStream at 5 minute and coarser granularities. The reason for the very different crossover point is the much larger degree of temporal locality in the cello99 trace.

5 Related Work

Although CDP has been gaining momentum in the industry and various products are available [7, 2, 4, 3] we are not aware of an enterprise-class storage controller with CDP support available as a product. There has been some research concerned with implementing CDP [23, 31, 14] but to the best of our knowledge we are the first to describe CDP architectures suitable for implementation in a high-end storage controller.

There has been some work examining every-write block based CDP with a focus on reducing space overhead [23, 31]. The paper describing the Peabody system makes a case for content-based coalescing of sectors: for the workloads they investigated, up to 84% of the written sectors have identical contents to sectors that were written previously and so could potentially be stored just once [23]. The paper describing the Trap-array system proposes to reduce the size of the CDP history by storing the compressed result of the XOR of each block with its previous version, instead of storing the block data. They find that it is common for only a small segment of a block to change between versions so that a XOR between the two versions yields a block containing mostly zeros the compresses well. Their results show up to two orders of magnitude space overhead reduction. The resulting cost is that retrieving a block version requires time proportional to the number of versions between the target version and the current version [31].

These results seem promising and the ideas presented in these papers can be considered complementary to our work. It may be interesting to investigate one of the proposed schemes as function of a variable protection granularity as neither paper does so.

The Clotho system is a Linux block device driver that supports creating an unlimited amount of read-only snapshots (versions) [14]. Since snapshot creation is efficient, frequent snapshots are feasible. Clotho is similar to the logging architecture in that read access to the current version requires metadata lookups. Unlike the logging architecture in Clotho access to newer versions is more efficient than access to historical versions. Similarly to the Trap-array system, a form of differential compression of extents with their previous versions is supported however its benefits on real-world workloads is not quantified.

Point-in-time volume snapshots [12] are a common feature supported by storage controllers, LVMs, filesystems and NAS boxes. We believe the results of our analysis of the checkpointing architecture are relevant for analysis of the overhead of periodic point-in-time snapshot support when this is implemented using COW (copy on write): on overwrite of the block on the production volume, the previous version of the block is copied to the snapshot volume. We presented results and analysis which provide insight into when COW-based architectures should be used and when alternatives should be considered as protection granularity and workload vary.

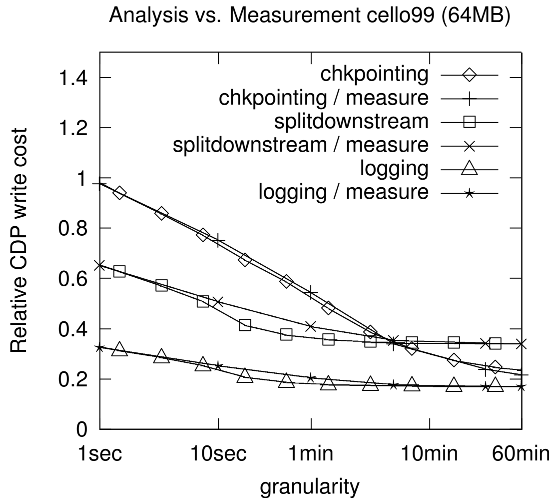

Figure 13: Tight match between analytical and empirical measurements of I/O counts for cello99 trace. The analytical calculation is according to the analysis developed in 2

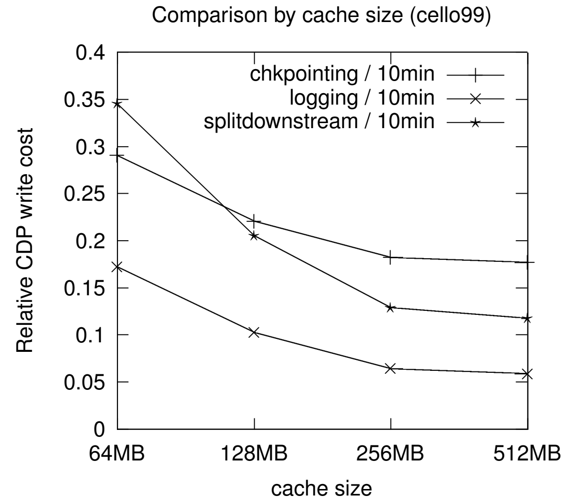

Figure 14: Write cache size versus relative CDP cost in terms of I/O's incurred by write requests for the 1 hour trace of the cello99 workload with a granularity of 10 minutes. As the cache size is increased, the cost reduces faster for the SplitDownStream architecture than for Checkpointing, and SplitDownStream dominates Checkpointing for cache sizes over 100MB. Note the exponential scale of the X-axis.

We are not aware of prior work which examined the issue from this perspective.

Network Appliance NAS filers and the Sun ZFS [11] filesystem support snapshotting of volumes by organizing the filesystem as a tree with user-data at the leaf level and meta-data held in internal nodes [19]. Each snapshot consists of a separate tree which may share subtrees with other snapshots. Creating a snapshot volume is done by creating a new tree root node which points to the same children as the original volume's root, so that the two volumes are represented by overlapping trees. Once a snapshot is created on a volume, any blocks which it shares with the snapshot (initially all blocks) cannot be updated in place, and all writes to them must be redirected. The first write to a block after a snapshot causes an entire tree path of meta-data to be allocated and copied, and linked to from the snapshot volume's tree root. Compared to in-place updates of meta-data, this approach seems to inherently require writing much more meta-data, especially for frequent snapshots. Some of this cost is balanced by delaying writes and then performing them in large sequential batches, similar to LFS [25]. However at high protection granularity it is not clear how competitive this architecture is. An investigation of the performance of this architecture as function of workload and protection granularity may be an interesting further work item.

Other work examined two options for implementing point-in-time snapshots, referred to as: COW and 'redirect on write' (ROW) [30]. The performance of a volume which has a single snapshot on it is examined and the two options are compared. No consideration is given to the cost related to periodically creating a new point-in-time copy. Very roughly the redirect-on-write architecture can be compared to the logging architecture while the COW architecture can be compared to our checkpointing architecture. The I/O cost per write for ROW is 1 I/O and, similarly to the case for logging, there is an extra meta-data lookup per read as well as a loss of spacial locality. The intuition that for write-dominated workloads ROW has an advantage while for read-dominated ones COW is advantageous is experimentally validated on various workloads. The impact of block size (minimal unit of copying to snapshot) on performance of the architectures is also investigated. Briefly, smaller block sizes benefit writes since less space is wasted while it hurts reads because of more fragmentation.

Versioned file systems [28, 27, 24, 29] keep track of updates to files and enable access to historical versions. The unit of protection in these file systems is a file - the user may access a historical version of a specific file - while in block-based CDP the unit of protection is an entire LUN. Some of this work discusses the overheads of meta-data related to versioning [24, 29] but our focus has been on the overheads associated with user data. The basic approach of investigating write related overheads as a function of protection granularity seems applicable to versioned file systems as well. One differ-

ence is that some versioned file systems support protection granularities that are not a constant amount of time, e.g. all the changes made to a file between the time it was opened by an application and the time it was closed may be considered to belong to a single version, whatever the amount of time this happens to take. This is similar to so called 'event-based' CDP where the user/application marks events of interest rather than deciding beforehand about protection granularity. Extending our model to handle this may be a further work item.

6 Conclusions

We proposed CDP architectures suitable for integration into storage controller, and analyzed them both analytically and empirically. Our analysis predicts the cost of each architecture in terms of write-related I/O and space overheads, and our empirical results confirm the analysis. Our work is the first to consider and accurately describe the effects of varying the CDP granularity. We show that one of the critical factors affecting both write I/O and space overheads is the fraction of retained writes which is determined by the relationship between the temporal distance of writes and the granularity. Workloads exhibiting high temporal locality w.r.t. the granularity perform well under the checkpointing architecture, whereas workloads exhibiting low temporal locality w.r.t. the granularity perform well under the SplitDownStream architecture. We analyzed specific workloads and showed that for the SPC1 Like OLTP workloads, a SplitDownStream architecture is superior for granularities up to 1 hour. We also showed that the Checkpointing architecture is superior for a workgroup file server workload such as cello99 for granularities coarser than 5 minutes. Aside from CDP, our results can also shed light on the performance overheads of common implementations of point-in-time copy in terms of the frequency of taking those copies.

6.1 Further Work

We do not claim definite conclusions regarding the cost of the CDP architectures for the general class of OLTP workloads or filesystem workloads. Our evaluation was not extensive enough to substantiate such claims. However we believe our results lay a foundation for a thorough investigation of real-world workloads.

A hybrid architecture which combines ideas from the Checkpointing architecture and the SplitDownStream architecture may offer the best of both: the behavior of SplitDownStream at higher granularities and the behavior of Checkpointing at lower granularities. Such an architecture is an interesting further work item.

Our evaluation did not consider I/O's related to metadata accesses to the LPMap CDP History structure. Also ignored in our evaluation is sequentiality of the write-related traffic. As described the checkpointing architecture requires 2 synchronous I/O's prior to cache destages of some pages. Evaluating the impact this has on performance (as well as attempting to overcome this limitation of the checkpointing architecture) is a further work item.

Acknowledgments

We would very much like to thank Uri Braun for his contribution at the early stages of our prototype, Chip Jarvis for discussions on versioning cache, Bruce McNutt for allowing us access to an earlier prototype implementation of the SPC-1 benchmark, HP for access to the cello99 traces, and Binny Gill and Dharmendra Modha for sending us their WOW prototype. Many thanks also to Kalman Meth, Sivan Tal, and the anonymous reviewers of the FAST program committee whose valuable feedback helped improve the paper.

References

[1] Continuous Data Protection: A Market Update, Byte and Switch Insider Report, July 2006. http://www.byteandswitch.com/insider.

[2] EMC Recover Point. http://www.emc.com/.

[3] FalconStor CDP. http://www.falconstor.com/.

[4] Mendocino Software, http://www.mendocinosoft.com/.

[5] Network Block Device. http://nbd.sourceforge.net/.

[6] Oracle Berkeley DB. http://www.oracle.com/database/berkeley-db/index.html.

[7] Revivio Inc. http://www.revivio.com/.

[8] Storage Performance Council, SPC-1 Benchmark Results. http://www.storageperformance.org/results.

[9] Storage Performance Council SPC-1 Specification. http://www.storageperformance.org/specs.

[10] The IBM TotalStorage DS8000 Series: Concepts and Architecture. IBM Redbook, 2005, http://www.redbooks.ibm.com/.

[11] ZFS: The last word in file systems. http://www.sun.com/2004-0914/feature/.

[12] A. Azagury, M. E. Factor, J. Satran, and W. Micka. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of IEEE/NASA Conf. Mass Storage Systems*, pages 259–270, 2002.

[13] J. Damoulakis. Continuous protection. *Storage, June 2004*, 3(4):33–39, 2004.

[14] M. Flouris and A. Bilas. Clotho: Transparent data versioning at the block I/O level. In *IEEE Symposium on Mass Storage Systems*, 2004.

[15] B. Gill and D. S. Modha. WOW: Wise ordering for writes combining spatial and temporal locality in non-volatile caches. In *Proceedings of USENIX File and Storage Technologies*, 2005.

[16] J. S. Glider, C. F. Fuente, and W. J. Scales. The software architecture of a SAN storage control system. *IBM Systems Journal*, 42(2):232–249, 2003.

[17] M. Hartung. IBM TotalStorage Enterprise Storage Server: A designer's view. *IBM Systems Journal*, 42(2):383–396, 2003.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; second edition*. Morgan Kaufmann, 1996.

[19] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter'94 USENIX Technical Conference*, pages 235–246, 1994.

[20] B. McNutt and S. A. Johnson. A standard test of I/O cache. In *Proc. of the Computer Measurements Group Conference*, 2001.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of USENIX File and Storage Technologies, San Francisco, CA.*, 2003.

[22] J. Menon. A performance comparison of RAID-5 and log-structured arrays. In *Fourth IEEE Symposium on High-Performance Distributed Computing*, 1995.

[23] C. B. Morrey III and D. Grunwald. Peabody: The time travelling disk. In *IEEE Symposium on Mass Storage Systems*, pages 241–253, 2003.

[24] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured filesystem. *ACM Transactions on Computer Systems*, pages 26–52, 1992.

[26] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the Winter USENIX Conference*, pages 405–420, 1993.

[27] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Otir. Deciding when to forget in the elephant file system. In *SOSP99, Symposium on Operating Systems Principles*, 1999.

[28] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmers workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, 1985.

[29] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of USENIX File And Storage Technologies*. USENIX, 2003.

[30] W. Xiao, Y. Liu, Q. K. Yang, J. Ren, and C. Xie. Implementation and performance evaluation of two snapshot methods on iSCSI target stores. In *Proceedings of IEEE/NASA Conf. Mass Storage Systems*, 2006.

[31] Q. Yang, W. Xiao, and J. Ren. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of International Symposium on Computer Architecture*, 2006.

[32] Y. Zhou and J. F. Philbin. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Annual Tech. Conf., Boston, MA*, pages 91–104, 2001.

Notes

[1] With the advent of RAID and volume virtualization, these are not necessarily true physical addresses.

[2] To calculate r, we only consider completed granularity windows, so that it becomes evident which writes are retained permanently. For "infinite" granularity, there is only one granularity window but it is never completed.

[3] We only consider those writes which have a subsequent write.

[4] The distribution of writes may not be as important if a versioning cache is used.

[5] Note that one cannot properly normalize according to the case of a regular volume (no CDP) because there are no retained writes, and there is no relationship between the size of the current stores in the various workloads.