Emulating Goliath Storage Systems with David

Nitin Agrawal[†], Leo Arulraj[*], Andrea C. Arpaci-Dusseau[*] and Remzi H. Arpaci-Dusseau[*]
NEC Laboratories America[†] University of Wisconsin–Madison[*]

Abstract

Benchmarking file and storage systems on *large* file-system images is important, but difficult and often infeasible. Typically, running benchmarks on such large disk setups is a frequent source of frustration for file-system evaluators; the scale alone acts as a strong deterrent against using larger albeit realistic benchmarks. To address this problem, we develop David: a system that makes it practical to run large benchmarks using modest amount of storage or memory capacities readily available on most computers. David creates a "compressed" version of the original file-system image by omitting all file data and laying out metadata more efficiently; an online storage model determines the runtime of the benchmark workload on the original uncompressed image. David works under any file system as demonstrated in this paper with ext3 and btrfs. We find that David reduces storage requirements by orders of magnitude; David is able to emulate a 1 TB target workload using only an 80 GB available disk, while still modeling the actual runtime accurately. David can also emulate newer or faster devices, *e.g.*, we show how David can effectively emulate a multi-disk RAID using a limited amount of memory on a desktop machine.

1 Introduction

File and storage systems are currently difficult to benchmark. Ideally, one would like to use a benchmark workload that is a realistic approximation of a known application. One would also like to run it in a configuration representative of real world scenarios, including realistic disk subsystems and file-system images.

In practice, realistic benchmarks and their realistic configurations tend to be much larger and more complex to set up than their trivial counterparts. File system traces (*e.g.*, from HP Labs [14]) are good examples of such workloads, often being large and unwieldy. Developing scalable yet practical benchmarks has long been a challenge for the storage systems community [12]. In particular, benchmarks such as GraySort [1] and SPECmail2009 [19] are compelling yet difficult to set up and use currently, requiring around 100 TB for GraySort and anywhere from 100 GB to 2 TB for SPECmail2009.

Benchmarking on large storage devices is thus a frequent source of frustration for file-system evaluators; the scale acts as a deterrent against using larger albeit realistic benchmarks [21], but running toy workloads on small disks is not sufficient. One obvious solution is to continually upgrade one's storage capacity. However, it is an expensive, and perhaps an infeasible solution to justify the costs and overheads solely for benchmarking.

Storage emulators such as Memulator [8] prove extremely useful for such scenarios – they let us prototype the "future" by pretending to plug in bigger, faster storage systems and run real workloads against them. Memulator, in fact, makes a strong case for storage emulation as *the* performance evaluation methodology of choice. But emulators are particularly tough: if they are to be big, they have to use existing storage (and thus are slow); if they are to be fast, they have to be run out of memory (and thus they are small).

The challenge we face is how can we get the best of both worlds? To address this problem, we have developed David, a "scale down" emulator that allows one to run large workloads by *scaling down* the storage requirements transparently to the workload. David makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system.

Our observation is that in many cases, the benchmark application does not care about the contents of individual files, but only about the structure and properties of the metadata that is being stored on disk. In particular, for the purposes of benchmarking, many applications do not write or read file contents at all (*e.g.*, `fsck`); the ones that do, often do not care what the contents are as long as *some* valid content is made available (*e.g.*, backup software). Since file data constitutes a significant fraction of the total file system size, ranging anywhere from 90 to 99% depending on the actual file-system image [2], avoiding the need to store file data has the potential to significantly reduce the required storage capacity during benchmarking.

The key idea in David is to create a "compressed" version of the original file-system image for the purposes of benchmarking. In the compressed image, unneeded user data blocks are omitted using novel classification techniques to distinguish data from metadata at scale; file system metadata blocks (e.g., inodes, directories and indirect blocks) are stored compactly on the available backing store. The primary benefit of the compressed image is to reduce the storage capacity required to run any given workload. To ensure that applications remain unaware of this interposition, whenever necessary, David synthetically generates file data on the fly; metadata I/O is redirected and accessed appropriately. David works under any file system; we demonstrate this using ext3 [22] and btrfs [23], two file systems very different in design.

Since David alters the original I/O patterns, it needs to model the runtime of the benchmark workload on the original uncompressed image. David uses an in-kernel model of the disk and storage stack to determine the run times of all individual requests as they would have executed on the uncompressed image. The model pays special attention to accurately modeling the I/O request queues; we find that modeling the request queues is crucial for overall accuracy, especially for applications issuing bursty I/O.

The primary mode of operation of David is the *timing-accurate* mode in which after modeling the runtime, an appropriate delay is inserted before returning to the application. A secondary *speedup* mode is also available in which the storage model returns instantaneously after computing the time taken to run the benchmark on the uncompressed disk; in this mode David offers the potential to reduce application runtime and speedup the benchmark itself. In this paper we discuss and evaluate David in the timing-accurate mode.

David allows one to run benchmark workloads that require file-system images orders of magnitude larger than the available backing store while still reporting the runtime as it would have taken on the original image. We demonstrate that David even enables emulation of faster and multi-disk systems like RAID using a small amount of memory. David can also aid in running large benchmarks on storage devices that are expensive or not even available in the market as it requires only a model of the non-existent storage device; for example, one can use a modified version of David to run benchmarks on a hypothetical 1TB SSD.

We believe David will be useful to file and storage developers, application developers, and users looking to benchmark these systems at scale. Developers often like to evaluate a prototype implementation at larger scales to identify performance bottlenecks, fine-tune optimizations, and make design decisions; analyses at scale often reveal interesting and critical insights into the system [12]. For example, how does one measure performance of a file system on a multi-disk multi-TB mirrored RAID configuration without having access to one? An end-user looking to select an application that works best at larger scale may also use David for emulation. For example, which anti-virus application scans a terabyte file system the fastest?

For workloads like PostMark, `mkfs`, `Filebench WebServer`, `Filebench VarMail`, and other microbenchmarks, we find that David delivers on its promise in reducing the required storage size while still accurately predicting the benchmark runtime for both ext3 and btrfs. Depending on the workload and the underlying file-system image, the size of the compressed image can range anywhere from 1 to 10% of the original, a large reduction in the required disk size for benchmarking. The storage model within David is fairly accurate in spite of operating in real-time within the kernel, and for most workloads predicts a runtime within 5% of the actual runtime. For example, for the Filebench webserver workload, David provides a 1000-fold reduction in required storage capacity and predicts a runtime within 0.08% of the actual.

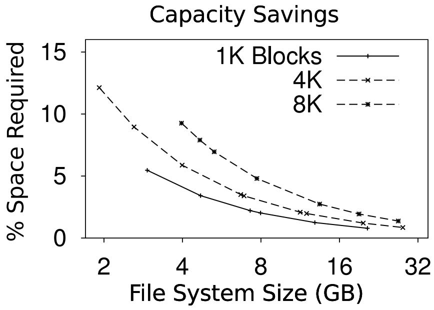

Figure 1: **Capacity Savings.** *Shows the savings in storage capacity if only metadata is stored, with varying file-size distribution modeled by (μ, σ) parameters of a lognormal distribution, (7.53, 2.48) and (8.33, 3.18) for the two extremes.*

2 David Overview

2.1 Extended Motivation

The motivating observation behind this work is that many benchmarks do not care about specific file contents but rather the overall properties of the file-system metadata. For such scenarios, if an evaluator can get away with storing only metadata blocks, there are significant gains to be made in the scale of the benchmark that can be run with the available storage capacity. The smaller the fraction of metadata blocks in any file-system image, the greater the advantages of storing only metadata.

One challenge in building David is how to deal with scale as we experiment with larger file systems containing many more files and directories. Given existing measurements of file systems, larger file systems actually contain larger files thereby requiring less space for metadata, a trend that favors David. Figure 1 shows the percentage of storage space occupied by metadata alone as compared to the total size of the file-system image written; the different file-system images for this experiment were generated by varying the file size distribution using the publicly available Impressions tool [3]. Using publicly available data on file-system metadata [2], we analyzed how file-size distribution changes with file systems of varying sizes. For example, how does the file-size distribution for 5 gigabyte file systems compare with 50 gigabyte ones?

We found that indeed larger file systems not only had more files, they also had larger files. For this experiment, the parameters of the lognormal distribution controlling the file sizes were changed along the x-axis to

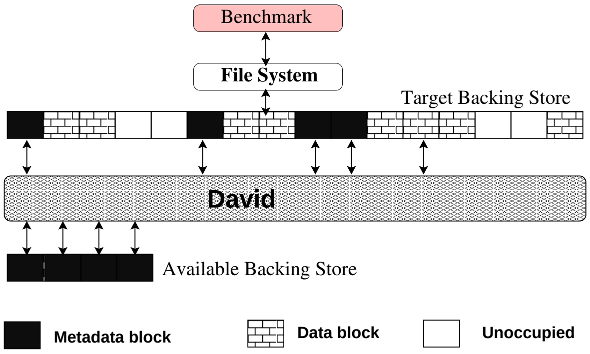

Figure 2: **Metadata Remapping and Data Squashing in David.** *The figure shows how metadata gets remapped and data blocks are squashed. The disk image above David is the* **target** *and the one below it is the* **available**.

generate progressively larger file systems with larger files therein. The relatively small fraction belonging to metadata (roughly 1 to 10%) as shown on the y-axis demonstrates the potential savings in storage capacity made possible if only metadata blocks are stored; David is designed to take advantage of this observation.

2.2 Design Goals for David

- **Scalability:** Emulating a large device requires David to maintain additional data structures and mimic several operations; our goal is to ensure that it works well as the underlying storage capacity scales.
- **Model accuracy:** An important goal is to model a storage device and accurately predict performance. The model should not only characterize the physical characteristics of the drive but also the interactions under different workload patterns.
- **Model overhead:** Equally important to being accurate is that the model imposes minimal overhead; since the model is inside the OS and runs concurrently with workload execution, it is required to be fairly fast.
- **Emulation flexibility:** David should be able to emulate different disks, storage subsystems, and multi-disk systems through appropriate use of backing stores.
- **Minimal application modification:** It should allow applications to run unmodified without knowing the significantly less capacity of the storage system underneath; modifications can be performed in limited cases only to improve ease of use but never as a necessity.

2.3 David Design

David exports a fake storage stack including a fake device of a much higher capacity than available. For the rest of the paper, we use the terms *target* to denote the hypothetical larger storage device, and *available* to denote the physically available system on which David is running, as shown in Figure 2. It also shows a schematic of how David makes use of metadata remapping and data squashing to free up a large percentage of the required storage space; a much smaller backing store can now service the requests of the benchmark.

David is implemented as a pseudo-device driver that is situated below the file system and above the backing store, interposing on all I/O requests. Since the driver appears as a regular device, a file system can be created and mounted on it. Being a loadable module, David can be used without any change to the application, file system or the kernel. Figure 3 presents the architecture of David with all the significant components and also shows the different types of requests that are handled within. We now describe the components of David.

First, the Block Classifier is responsible for classifying blocks addressed in a request as data or metadata and preventing I/O requests to data blocks from going to the backing store. David intercepts all writes to data blocks, records the block address if necessary, and discards the actual write using the Data Squasher. I/O requests to metadata blocks are passed on to the *Metadata Remapper*.

Second, the Metadata Remapper is responsible for laying out metadata blocks more efficiently on the backing store. It intercepts all write requests to metadata blocks, generates a remapping for the set of blocks addressed, and writes out the metadata blocks to the remapped locations. The remapping is stored in the Metadata Remapper to service subsequent reads.

Third, writes to data blocks are not saved, but reads to these blocks could still be issued by the application; in order to allow applications to run transparently, the Data Generator is responsible for generating synthetic content to service subsequent reads to data blocks that were written earlier and discarded. The Data Generator contains a number of built-in schemes to generate different kinds of content and also allows the application to provide hints to generate more tailored content (*e.g.*, binary files).

Finally, by performing the above-mentioned tasks David modifies the original I/O request stream. These modifications in the I/O traffic substantially change the application runtime rendering it useless for benchmarking. The Storage Model in David carefully models the (potentially different) target storage subsystem underneath to predict the benchmark runtime on the target system. By doing so in an online fashion with little overhead, the Storage Model makes it feasible to run large workloads in a space and time-efficient manner. The individual components are discussed in detail in §3 through §6.

2.4 Choice of Available Backing Store

David is largely agnostic to the choice of the backing store for available storage: HDDs, SSDs, or memory can be used depending on the performance and capacity requirements of the target device being emulated. Through a significant reduction in the number of device I/Os,

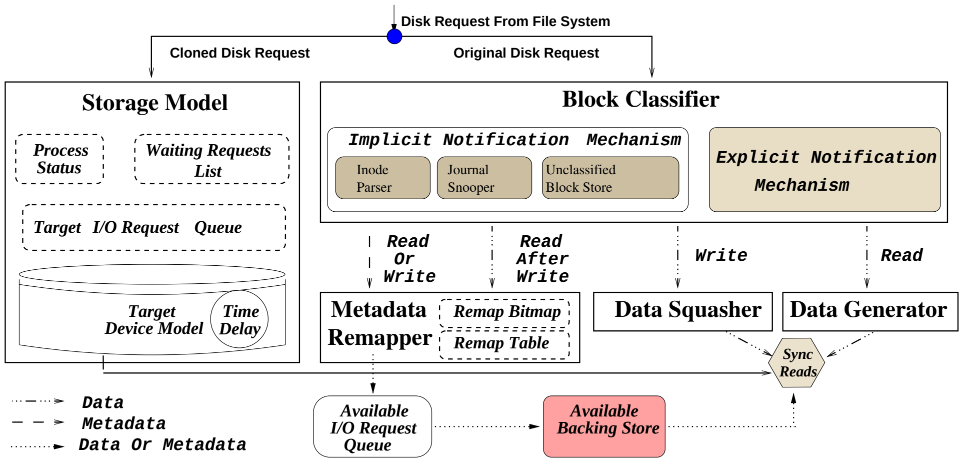

Figure 3: **David Architecture.** *Shows the components of David and the flow of requests handled within.*

David compensates for its internal book-keeping overhead and also for small mismatches between the emulated and available device. However, if one wishes to emulate a device much faster than the available device, using memory is a safer option. For example, as shown in §6.3, David successfully emulates a RAID-1 configuration using a limited amount of memory. If the performance mismatch is not significant, a hard disk as backing store provides much greater scale in terms of storage capacity. Throughout the paper, "available storage" refers to the backing store in a generic sense.

2.5 Alternate Approaches

Memulator [8] makes a great case for why storage emulation provides the unique ability to explore nonexistent storage components and take end-to-end measurements. While Memulator focuses on the diversity of devices being emulated, David focuses on the scalability of the emulation itself. We now briefly discuss some alternatives.

One alternate is to simply buy a larger capacity disk or a newer device and use it to run the benchmarks. This is sometimes feasible, but often not desirable. Even if one buys a larger disk, in the future they would need an even larger one; David allows one to keep up with this arms race without always investing in new devices. Note that we chose 1 TB as the upper limit for evaluation in this paper because we could validate our results for that size. Having a large disk will also not address the issue of emulating much faster devices such as SSDs or RAID configurations. David emulates faster devices through an efficient use of memory as backing store.

Another alternate is to simulate the storage component under test; disk simulators like Disksim [5] allow such an evaluation flexibly. However, simulation results are often far from perfect [7] – they fail to capture system dependencies and require the generation of representative I/O traces, another challenge faced by the storage community.

Finally, one might use pure analytical modeling for the storage devices; while very useful in some circumstances, it is not without its own set of challenges and limitations [17]. In particular, it is extremely hard to capture the interactions and complexities in real systems. Wherever possible, David does leverage well-tested analytical models for individual components to aid the emulation.

Both simulation and analytical modeling are complementary to emulation, perfectly useful in their own right. Emulation does however provide a reasonable middle ground in terms of flexibility and realism.

3 Block Classification

The primary requirement for David to prevent data writes using the Data Squasher is the ability to classify a block as metadata or data. David provides both implicit and explicit block classification. The implicit approach is more laborious but provides a flexible approach to run unmodified applications and file systems. The explicit notification approach is straightforward and much simpler to implement, albeit at the cost of a small modification in the operating system or the application; both are available in David and can be chosen according to the requirements of the evaluator. The implicit approach is demonstrated using ext3 and the explicit approach using btrfs.

3.1 Implicit Type Detection

For ext2 and ext3, the majority of the blocks are statically assigned for a given file system size and configuration at the time of file system creation; the allocation for these blocks doesn't change during the lifetime of the file system. Blocks that fall in this category include the super block, group descriptors, inode and data bitmaps, inode blocks and blocks belonging to the journal; these blocks are relatively straightforward to classify based on their on-disk location, or their Logical Block Address (LBA). However, not all blocks are statically assigned; dynamically-allocated blocks include directory, indirect (single, double, or triple indirect) and data blocks. Unless all blocks contain some self-identification information, in order to accurately classify a dynamically allocated block, the system needs to track the inode pointing to the particular block to infer its current status.

Implicit classification is based on prior work on Semantically-Smart Disk Systems (SDS) [18], but there are two significant challenges David must address. First, as opposed to SDS, David has to ensure that no metadata blocks are ever misclassified. Second, benchmark scalability introduces additional memory pressure to handle delayed classification. In this paper we only discuss our new contributions (the original SDS paper provides details of the basic block-classification mechanisms).

3.1.1 Unclassified Block Store

To infer when a file or directory is allocated and deallocated, David tracks writes to inode blocks, inode bitmaps and data bitmaps; to enumerate the indirect and directory blocks that belong to a particular file or directory, it uses the contents of the inode. It is often the case that the blocks pointed to by an inode are written out before the corresponding inode block; if a classification attempt is made when a block is being written, an indirect or directory block will be misclassified as an ordinary data block. This transient error is unacceptable for David since it leads to the "metadata" block being discarded prematurely and could cause irreparable damage to the file system. For example, if a directory or indirect block is accidentally discarded, it could lead to file system corruption.

To rectify this problem, David temporarily buffers in memory writes to all blocks which are as yet unclassified, inside the *Unclassified Block Store* (UBS). These write requests remain in the UBS until a classification is made possible upon the write of the corresponding inode. When a corresponding inode does get written, blocks that are classified as metadata are passed on to the Metadata Remapper for remapping; they are then written out to persistent storage at the remapped location. Blocks classified as data are discarded at that time. All entries in the UBS corresponding to that inode are also removed.

The UBS is implemented as a list of block I/O (*bio*) request structures. An extra reference to the memory pages pointed to by these bio structures is held by David as long they remain in the UBS; this reference ensures that these pages are not mistakenly freed until the UBS is able to classify and persist them on disk, if needed. In order to reduce the inode parsing overhead otherwise imposed for each inode write, David maintains a list of recently written inode blocks that need to be processed and uses a separate kernel thread for parsing.

3.1.2 Journal Snooping

Storing unclassified blocks in the UBS can cause a strain on available memory in certain situations. In particular, when ext3 is mounted on top of David in ordered journaling mode, all the data blocks are written to disk at journal-commit time but the metadata blocks are written to disk only at the checkpoint time which occurs much less frequently. This results in a temporary yet precarious build up of data blocks in the UBS even though they are bound to be squashed as soon as the corresponding inode is written; this situation is especially true when large files (*e.g.*, 10s of GB) are written. In order to ensure the overall scalability of David, handling large files and the consequent explosion in memory consumption is critical.

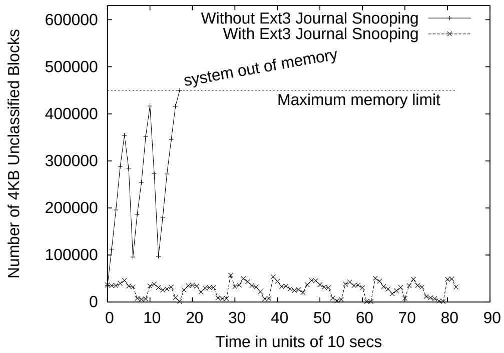

Figure 4: **Journal Snooping's effect on memory usage.**

To reduce the memory pressure under this scenario, David snoops on the journal commit traffic for inodes and indirect blocks logged within a committed transaction; this enables block classification even prior to checkpoint. When a journal-descriptor block is written as part of a transaction, David records the blocks that are being logged within that particular transaction. In addition, all journal writes within that transaction are cached in memory until the transaction is committed. After that, the inodes and their corresponding direct and indirect blocks are processed to allow block classification; the identified data blocks are squashed from the UBS and the identified metadata blocks are remapped and stored persistently.

Figure 4 compares the memory pressure with and without Journal Snooping demonstrating its effectiveness. It shows the number of 4 KB block I/O requests resident in the UBS sampled at 10 sec intervals during the creation of a 24 GB file on ext3; the file system is mounted on top of David in ordered journaling mode with a commit interval of 5 secs. This experiment was run on a dual core machine with 2 GB memory. Since this workload is data write intensive, without Journal Snooping, the system runs out of memory when around 450,000 bio requests are in the UBS (occupying roughly 1.8 GB of memory). Journal Snooping ensures that the memory consumed by outstanding bio requests does not go beyond a maximum of 240 MB.

3.2 Explicit Metadata Notification

David is meant to be useful for a wide variety of file systems; explicit metadata notification provides a mechanism to rapidly adopt a file system for use with David. Since data writes can come only from the benchmark application in user-space whereas metadata writes are issued by the file system, our approach is to identify the data blocks before they are even written to the file system. Our implementation of explicit notification is thus file-system agnostic – it relies on a small modification to the page cache to collect additional information. We demonstrate the benefits of this approach using btrfs, a file system quite unlike ext3 in design.

When an application writes to a file, David captures the pointers to the in-memory pages where the data content is stored, as it is being copied into the page cache. Subsequently, when the writes reach David, they are compared against the captured pointer addresses to decide whether the write is to metadata or data. Once the presence is tested, the pointer is removed from the list since the same page can be reused for metadata writes in the future.

There are certainly other ways to implement explicit notification. Instead of identifying data blocks with the page cache modification, one can modify the file system to explicitly flag the metadata blocks. We believe our approach is easier to implement, does not require any file system modification, and is also easier to extend to RAID since parity blocks are automatically classified as metadata and not discarded.

4 Metadata Remapping

Since David exports a target pseudo device of much higher capacity to the file system than the available storage device, the bio requests issued to the pseudo device will have addresses in the full target range and thus need to be suitably remapped. For this purpose, David maintains a remap table called Metadata Remapper which maps "target" addresses to "available" addresses. The Metadata Remapper can contain an entry either for one metadata block (*e.g.*, super block), or a range of metadata blocks (*e.g.*, group descriptors); by allowing an arbitrary range of blocks to be remapped together, the Metadata Remapper provides an efficient translation service that also provides scalability. Range remapping in addition preserves sequentiality of the blocks if a disk is used as the backing store. In addition to the Metadata Remapper, a *remap bitmap* is maintained to keep track of free and used blocks on the available physical device; the remap bitmap supports allocation both of a single remapped block and a range of remapped blocks.

The destination (or remapped) location for a request is determined using a simple algorithm which takes as input the number of contiguous blocks that need to be remapped and finds the first available chunk of space from the *remap bitmap*. This can be done statically or at runtime; for the ext3 file system, since most of the blocks are statically allocated, the remapping for these blocks can also be done statically to improve performance. Subsequent writes to other metadata blocks are remapped dynamically; when metadata blocks are deallocated, corresponding entries from the Metadata Remapper and the remap bitmap are removed. From our experience, this simple algorithm lays out blocks on disk quite efficiently. More sophisticated allocation algorithms based on locality of reference can be implemented in the future, if needed.

5 Data Generator

David services the requirements of systems oblivious to file content with data squashing and metadata remapping. However, many real applications care about file content; the Data Generator with David is responsible for generating synthetic content to service read requests to data blocks that were previously discarded.

Many systems that read back previously written data do not care about the *specific* content within the files as long as there is *some* content (*e.g.*, a file-system backup utility, or the Postmark benchmark). Much in the same way as failure-oblivious computing generates values to service reads to invalid memory while ignoring invalid writes [15], David randomly generates content to service out-of-bound read requests.

Some systems may expect file contents to have valid syntax or semantics; the performance of these systems depend on the actual content being read (*e.g.*, a desktop search engine for a file system, or a spell-checker). For such systems, naive content generation would either crash the application or give poor benchmarking results. David produces valid file content leveraging prior work on generating file-system images [3].

Finally, some systems may expect to read back data exactly as they wrote earlier (*i.e.*, a read-after-write or RAW dependency) or expect a precise structure that cannot be generated arbitrarily (*e.g.*, a binary file or a configuration file). David provides additional support to run these demanding applications using the *RAW Store*, designed as a cooperative resource visible to the user and configurable to suit the needs of different applications.

Our current implementation of RAW Store is very simple: in order to decide which data blocks need to be stored persistently, David requires the application to supply a list of the relevant file paths. David then looks up the inode number of the files and tracks all data blocks pointed to by these inodes, writing them out to disk using the Metadata Remapper just as any metadata block. In the future, we intend to support more nuanced ways to maintain the RAW Store; for example, specifying directories instead of files, or by using *Memoization* [11].

For applications that must exactly read back a significant fraction of what they write, the scalability advantage of David diminishes; in such cases the benefits are primarily from the ability to emulate new devices.

Parameter	H1	H2	Parameter	H1	H2
Disk size	80 GB	1 TB	Cache segments	11	500
Rotational Speed	7200 RPM	7200 RPM	Cache R/W partition	Varies	Varies
Number of cylinders	88283	147583	Bus Transfer	133 MBps	133 MBps
Number of zones	30	30	Seek profile(long)	3800+(cyl*116)/10^3	3300+(cyl*5)/10^6
Sectors per track	567 to 1170	840 to 1680	Seek profile(short)	300+$\sqrt{(cyl * 2235)}$	700+$\sqrt{cyl}$
Cylinders per zone	1444 to 1521	1279 to 8320	Head switch	1.4 ms	1.4 ms
On-disk cache size	2870 KB	300 MB	Cylinder switch	1.6 ms	1.6 ms
Disk cache segment	260 KB	600 KB	Dev driver req queue*	128-160	128-160
Req scheduling*	FIFO	FIFO	Req queue timeout*	3 ms (unplug)	3 ms (unplug)

Table 1: **Storage Model Parameters in David.** *Lists important parameters obtained to model disks Hitachi HDS728080PLA380 (H1) and Hitachi HDS721010KLA330 (H2). * denotes parameters of I/O request queue (IORQ).*

6 Storage Model and Emulation

Not having access to the target storage system requires David to precisely capture the behavior of the entire storage stack with all its dependencies through a model. The storage system modeled by David is the *target* system and the system on which it runs is the *available* system. David emulates the behavior of the target disk by sending requests to the available disk (for persistence) while simultaneously sending the *target request stream* to the Storage Model; the model computes the time that would have taken for the request to finish on the target system and introduces an appropriate delay in the actual request stream before returning control. Figure 3 presented in §2 shows this setup more clearly.

As a general design principle, to support low-overhead modeling without compromising accuracy, we avoid using any technique that either relies on storing empirical data to compute statistics or requires table-based approaches to predict performance [4]; the overheads for such methods are directly proportional to the amount of runtime statistics being maintained which in turn depends on the size of the disk. Instead, wherever applicable, we have adopted and developed analytical approximations that did not slow the system down; our resulting models are sufficiently lean while still being fairly accurate.

To ensure portability of our models, we have refrained from making device-specific optimizations to improve accuracy; we believe current models in David are fairly accurate. The models are also adaptive enough to be easily configured for changes in disk drives and other parameters of the storage stack. We next present some details of the disk model and the storage stack model.

6.1 Disk Model

David's disk model is based on the classical model proposed by Ruemmler and Wilkes [16], henceforth referred as the RW model. The disk model contains information about the disk geometry (*i.e.*, platters, cylinders and zones) and maintains the current disk head position; using these sources it models the disk seek, rotation, and transfer times for any request. The disk model also keeps track of the effects of disk caches (track prefetching, write-through and write-back caches). In the future, it will be interesting to explore using Disksim for the disk model; we will need to ensure it does not appreciably slow down the emulation when used without memory as backing store.

6.1.1 Disk Drive Profile

The disk model requires a number of drive-specific parameters as input, a list of which is presented in the first column of Table 1; currently David contains models for two disks: the Hitachi HDS728080PLA380 80 GB disk, and the Hitachi HDS721010KLA330 1 TB disk. We have verified the parameter values for both these disks through carefully controlled experiments. David is envisioned for use in environments where the target drive itself may not be available; if users need to model additional drives, they need to supply the relevant parameters. Disk seeks, rotation time and transfer times are modeled much in the same way as proposed in the RW model. The actual parameter values defining the above properties are specific to a drive; empirically obtained values for the two disks we model are shown in Table 1.

6.1.2 Disk Cache Modeling

The drive cache is usually small (few hundred KB to a few MB) and serves to cache reads from the disk media to service future reads, or to buffer writes. Unfortunately, the drive cache is one of the least specified components as well; the cache management logic is low-level firmware code which is not easy to model.

David models the number and size of segments in the disk cache and the number of disk sector-sized slots in each segment. Partitioning of the cache segments into read and write caches, if any, is also part of the information contained in the disk model. David models the read cache with a FIFO eviction policy. To model the effects of write caching, the disk model maintains statistics on the current size of writes pending in the disk cache and the time needed to flush these writes out to the media. Write buffering is simulated by periodically emptying a fraction of the contents of the write cache during idle periods of the disk in between successive foreground requests. The cache is modeled with a write-through policy and is partitioned into a sufficiently large read cache to match the read-ahead behavior of the disk drive.

6.2 Storage Stack Model

David also models the I/O request queues (IORQs) maintained in the OS; Table 1 lists a few of its important parameters. While developing the Storage Model, we found that accurately modeling the behavior of the IORQs is crucial to predict the target execution time correctly. The IORQs usually have a limit on the maximum number of requests that can be held at any point; processes that try to issue an I/O request when the IORQ is full are made to wait. Such waiting processes are woken up when an I/O issued to the disk drive completes, thereby creating an empty slot in the IORQ. Once woken up, the process is also granted privilege to batch a constant number of additional I/O requests even when the IORQ is full, as long as the total number of requests is within a specified upper limit. Therefore, for applications issuing bursty I/O, the time spent by a request in the IORQ can outweigh the time spent at the disk by several orders of magnitude; modeling the IORQs is thus crucial for overall accuracy.

Disk requests arriving at David are first enqueued into a *replica queue* maintained inside the Storage Model. While being enqueued, the disk request is also checked for a possible *merge* with other pending requests: a common optimization that reduces the number of total requests issued to the device. There is a limit on the number of disk requests that can be merged into a single disk request; eventually merged disk requests are dequeued from the *replica queue* and dispatched to the disk model to obtain the service time spent at the drive. The *replica queue* uses the same request scheduling policy as the target IORQ.

6.3 RAID Emulation

David can also provide effective RAID emulation; each disk in a RAID configuration is emulated using a memory-backed "compressed" device underneath software RAID. David exports multiple block devices with separate major and minor numbers; it differentiates requests to different devices using the major number. For the purpose of performance benchmarking, David uses a single memory-based backing store for all the compressed RAID devices. Using multiple threads, the Storage Model maintains separate state for each of the devices being emulated. Requests are placed in a single request queue tagged with a device identifier; individual Storage Model threads for each device fetch one request at a time from this request queue based on the device identifier. Similar to the single device case, the servicing thread calculates the time at which a request to the device should finish and notifies completion using a callback.

7 Evaluation

We seek to answer four important questions. First, what is the accuracy of the Storage Model? Second, how accurately does David predict benchmark runtime and what storage space savings does it provide? Third, can David scale to large target devices including RAID? Finally, what is the memory and CPU overhead of David?

7.1 Experimental Platform

We have developed David for the Linux operating system. The hard disks currently modeled are the 1 TB Hitachi HDS721010KLA330 (referred to as D_{1TB}) and the 80 GB Hitachi HDS728080PLA380 (referred to as D_{80GB}); table 1 lists their relevant parameters. Unless specified otherwise, the following hold for all the experiments: (1) machine used has a quad-core Intel processor and 4GB RAM running Linux 2.6.23.1 (2) ext3 file system is mounted in ordered-journaling mode with a commit interval of 5 sec (3) microbenchmarks were run directly on the disk without a file system (4) David predicts the benchmark runtime for a target D_{1TB} while in fact running on the available D_{80GB} (5) to validate accuracy, David was instead run directly on D_{1TB}.

7.2 Storage Model Accuracy

First, we validate the accuracy of Storage Model in predicting the benchmark runtime on the target system. Since our aim is to validate the accuracy of the Storage Model alone, we run David in a *model only* mode where we disable block classification, remapping and data squashing. David just passes down the requests that it receives to the available request queue below. We run David on top of D_{1TB} and set the target drive to be the same. Note that the available system is the *same* as the target system for these experiments since we only want to compare the measured and modeled times to validate the accuracy of the Storage Model. Each block request is traced along its path from David to the disk drive and back. This is done in order to measure the total time that the request spends in the available IORQ and the time spent getting serviced at the available disk. These measured times are then compared with the modeled times obtained from the Storage Model.

Figure 5 shows the Storage Model accuracy for four micro-workloads: sequential and random reads, and sequential and random writes; these micro-workloads have demerit figures of 24.39, 5.51, 0.08, and 0.02 respectively, as computed using the Ruemmler and Wilkes methodology [16]. The large demerit for sequential reads is due to a variance in the available disk's cache-read times; modeling the disk cache in greater detail in the future could potentially avoid this situation. However, sequential read requests do not contribute to a measurably large error in the total modeled runtime; they often hit the disk cache and have service times less than 500 microseconds while other types of disk requests take around 20 to 35 milliseconds to get serviced. Any inaccuracy in the modeled times for sequential read requests are negligible when compared to the service times of other types of disk requests. Thus,

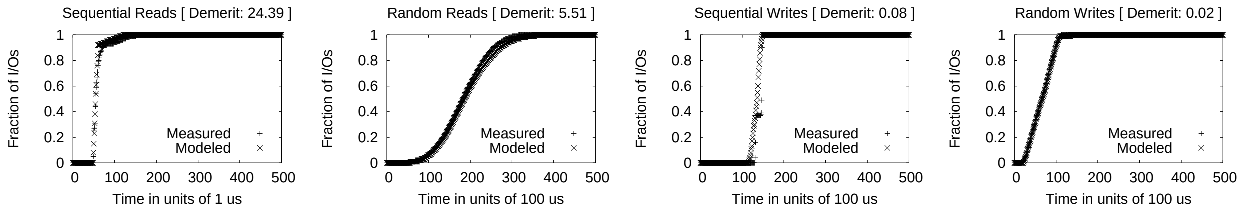

Figure 5: **Storage Model accuracy for Sequential and Random Reads and Writes.** *The graph shows the cumulative distribution of measured and modeled times for sequential and random reads and writes.*

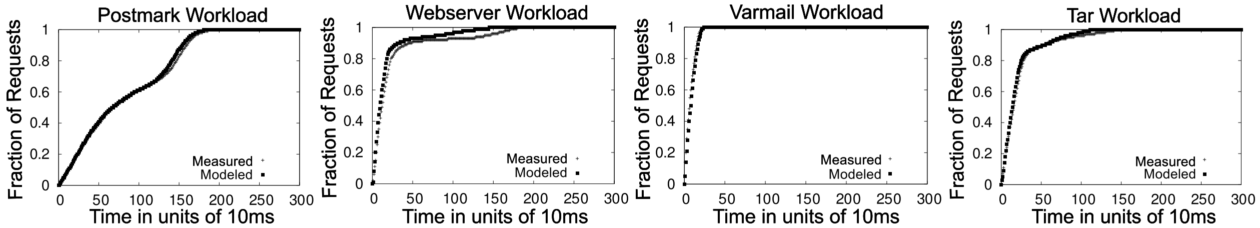

Figure 6: **Storage Model accuracy.** *The graphs show the cumulative distribution of measured and modeled times for the following workloads from left to right: Postmark, Webserver, Varmail and Tar.*

	Implicit Classification – Ext3						Explicit Notification – Btrfs		
Benchmark Workload	Original Storage (KB)	David Storage (KB)	Storage Savings (%)	Original Runtime (Secs)	David Runtime (Secs)	Runtime Error (%)	Original Runtime (Secs)	David Runtime (Secs)	Runtime Error (%)
mkfs	976762584	7900712	99.19	278.66	281.81	1.13	-	-	-
imp	11224140	18368	99.84	344.18	339.42	-1.38	327.294	324.057	0.99
tar	21144	628	97.03	257.66	255.33	-0.9	146.472	135.014	7.8
grep	-	-	-	250.52	254.40	1.55	141.960	138.455	2.47
virus scan	-	-	-	55.60	47.95	-13.75	27.420	31.555	15.08
find	-	-	-	26.21	26.60	1.5	-	-	-
du	-	-	-	102.69	101.36	-1.29	-	-	-
postmark	204572	404	99.80	33.23	29.34	-11.69	22.709	22.243	2.05
webserver	3854828	3920	99.89	127.04	126.94	-0.08	125.611	126.504	0.71
varmail	7852	3920	50.07	126.66	126.27	-0.31	126.019	126.478	0.36
sr	-	-	-	40.32	44.90	11.34	40.32	44.90	11.34
rr	-	-	-	913.10	935.46	2.45	913.10	935.46	2.45
sw	-	-	-	57.28	58.96	2.93	57.28	58.96	2.93
rw	-	-	-	308.74	291.40	-5.62	308.74	291.40	-5.62

Table 2: **David Performance and Accuracy.** *Shows savings in capacity, accuracy of runtime prediction, and the overhead of storage modeling for different workloads. Webserver and varmail are generated using FileBench; virus scan using AVG.*

we decided to not make the disk-cache model any more complex for the sake of sequential reads.

Figure 6 shows the accuracy for four different macro workloads and application kernels: Postmark [10], webserver (generated using FileBench [13]), Varmail (mail server workload using FileBench), and a Tar workload (copy and untar of the linux kernel of size 46 MB).

The FileBench Varmail workload emulates an NFS mail server, similar to Postmark, but is multi-threaded instead. The Varmail workload consists of a set of open/read/close, open/append/close and deletes in a single directory, in a multi-threaded fashion. The FileBench webserver workload comprises of a mix of open/read/close of multiple files in a directory tree. In addition, to simulate a webserver log, a file append operation is also issued. The workload consists of 100 threads issuing 16 KB appends to the weblog every 10 reads.

Overall, we find that storage modeling inside David is quite accurate for all workloads used in our evaluation. The total modeled time as well as the distribution of the individual request times are close to the total measured time and the distribution of the measured request times.

7.3 David Accuracy

Next, we want to measure how accurately David predicts the benchmark runtime. Table 2 lists the accuracy and storage space savings provided by David for a variety of benchmark applications for both ext3 and btrfs. We have chosen a set of benchmarks that are commonly used and also stress various paths that disk requests take within David. The first and second columns of the table show the storage space consumed by the benchmark workload without and with David. The third column shows the percentage savings in storage space achieved by using David. The fourth column shows the original benchmark runtime without David on D_{1TB}. The fifth column shows the benchmark runtime with David on D_{80GB}. The sixth column shows the percentage error in the prediction of the benchmark runtime by David. The final three columns show the original and modeled runtime, and the percentage error for the btrfs experiments; the storage space savings are roughly the same as for ext3. The *sr*, *rr*, *sw*, and *rw* workloads are run directly on the raw device and hence are independent of the file system.

mkfs creates a file system with a 4 KB block size over the 1 TB target device exported by David. This workload only writes metadata and David remaps writes issued by *mkfs* sequentially starting from the beginning of D_{80GB}; no data squashing occurs in this experiment.

imp creates a realistic file-system image of size 10 GB using the publicly available Impressions tool [3]. A total of 5000 regular files and 1000 directories are created with an average of 10.2 files per directory. This workload is a data-write intensive workload and most of the issued writes end up being squashed by David.

tar uses the GNU tar utility to create a gzipped archive of the file-system image of size 10 GB created by *imp*; it writes the newly created archive in the same file system. This workload is a data read and data write intensive workload. The data reads are satisfied by the Data Generator without accessing the available disk, while the data writes end up being squashed.

grep uses the GNU grep utility to search for the expression "nothing" in the content generated by both *imp* and *tar*. This workload issues significant amounts of data reads and small amounts of metadata reads. *virus scan* runs the AVG virus scanner on the file-system image created by *imp*. *find* and *du* run the GNU find and GNU du utilities over the content generated by both *imp* and *tar*. These two workloads are metadata read only workloads.

David works well under both the implicit and explicit approaches demonstrating its usefulness across file systems. Table 2 shows how David provides tremendous savings in the required storage capacity, upwards of 99% (a 100-fold or more reduction) for most workloads. David also predicts benchmark runtime quite accurately. Prediction error for most workloads is less than 3%, although for a few it is just over 10%.

7.4 David Scalability

David is aimed at providing scalable emulation using commodity hardware; it is important that accuracy is not compromised at larger scale. Figure 7 shows the accuracy and storage space savings provided by David while creating file-system images of 100s of GB. Using an available capacity of only 10 GB, David can model the runtime of Impressions in creating a realistic file-system image of 800 GB; in contrast to the linear scaling of the target capacity demanded, David barely requires any extra available capacity. David also predicts the benchmark runtime within a maximum of 2.5% error even with the huge disparity between target and available disks at the 800 GB mark, as shown in Figure 7.

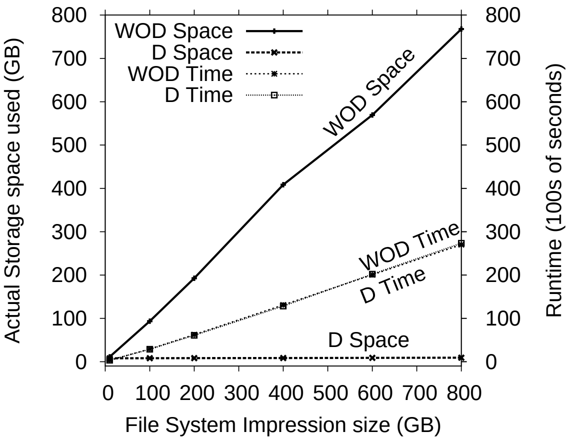

Figure 7: **Storage Space Savings and Model Accuracy.** *The "Space" lines show the savings in storage space achieved when using David for the* impressions *workload with file-system images of varying sizes until 800GB; "Time" lines show the accuracy of runtime prediction for the same workload.* **WOD***: space/time without David,* **D***: space/time with David.*

Num Disks	Rand W (s)	Rand R (s)
1*	48.108	49.802
2	48.060	23.552
3	48.039	15.840
4	48.130	11.922

Table 3: **David RAID Emulation.** *Shows time taken in seconds for a RAID-1 setup using David with memory as backing store; workload issues 20000 read and write requests through 2 concurrent processes.* * *denotes 1 disk with David w/o RAID.*

The reason we limit these experiments to a target capacity of less than 1 TB is because we had access to only a terabyte sized disk against which we could validate the accuracy of David. Extrapolating from this experience, we believe David will enable one to emulate disks of 10s or 100s of TB given the 1 TB disk.

7.5 David for RAID

Due to lack of space we present only a single set of experiments for emulating RAID using David. Table 3 shows a simple experiment where David emulates a multi-disk RAID-1 (mirrored) configuration; each device is emulated using a memory-disk as backing store. However, since the

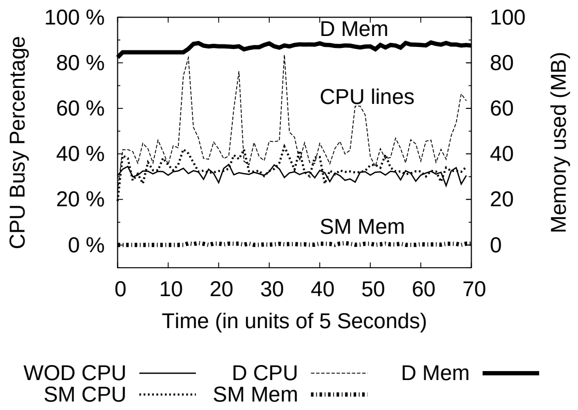

Figure 8: **David CPU and Memory Overhead.** *Shows the memory and percentage CPU consumption by David while creating a 10 GB file-system image using* impressions. **WOD CPU***: CPU without David,* **SM CPU***: CPU with Storage Model alone,* **D CPU***: total CPU with David,* **SM Mem***: Storage Model memory alone,* **D Mem***: total memory with David.*

multiple disks contain copies of the same block, a single physical copy is stored, further reducing the memory footprint. The first line shows the baseline performance using David with 1 disk but without RAID. As expected, the performance of random writes remains roughly the same as the number of disks is increased. Random read performance improves as expected too; for example, when the number of disks increases from 1 to 2, the performance improves two-fold. In this paper we do not present verification of the RAID results against real hardware; it is part of our on-going work.

7.6 David Overhead

David is designed to be used for benchmarking and not as a production system, thus scalability and accuracy are the more relevant metrics of evaluation; we do however want to measure the memory and CPU overhead of using David on the available system to ensure it is practical to use. All memory usage within David is tracked using several counters; David provides support to measure the memory usage of its different components using `ioctls`. To measure the CPU overhead of the Storage Model alone, David is run in the *model-only* mode where block classification, remapping and data squashing are turned off.

In our experience with running different workloads, we found that the memory and CPU usage of David is acceptable for the purposes of benchmarking. As an example, Figure 8 shows the CPU and memory consumption by David captured at 5 second intervals while creating a 10 GB file-system image using Impressions. For this experiment, the Storage Model consumes less than 1 MB of memory; the average memory consumed in total by David is less than 90 MB, of which the pre-allocated cache used by the Journal Snooping to temporarily store the journal writes itself contributes 80 MB. Amount of CPU used by the Storage Model alone is insignificant, however implicit classification by the Block Classifier is the primary consumer of CPU using 10% on average with occasional spikes. The CPU overhead is not an issue at all if one uses the explicit notification approach instead.

8 Related Work

Evaluation of how well an I/O system scales has been of interest in prior research and is becoming increasingly more relevant. Chen and Patterson proposed a "self-scaling" benchmark that scales with the I/O system being evaluated, to stress the system in meaningful ways [6]. Although useful for disk and I/O systems, the self-scaling benchmarks are not directly applicable for file systems. The evaluation of the XFS file system from Silicon Graphics uses a number of benchmarks specifically intended to test its scalability [20]; such an evaluation can benefit from David to employ even larger benchmarks with greater ease; SpecSFS [24] also contains some techniques for scalable workload generation. At a recent storage benchmarking workshop, both academia and industry participants agreed on the need to run scalable benchmarks as an important requirement for storage benchmarking [25]; David provides a platform to do so with ease.

Similar to our emulation of scale in a storage system, Gupta *et al.* from UCSD propose a technique called *time dilation* for emulating network speeds orders of magnitude faster than available [9]. Time dilation allows one to experiment with unmodified applications running on commodity operating systems by subjecting them to much faster network speeds than actually available.

Memulator is a "timing-accurate" storage emulator from CMU [8]; timing-accurate emulation allows a simulated storage component to be plugged into a real system running real applications. Memulator can use the memory of either a networked machine or the local machine as the storage media of the emulated disk, enabling full system evaluation of hypothetical storage devices; David can benefit from the networked-emulation capabilities of Memulator in scenarios when either the host machine has limited CPU and memory resources, or when the interference of running David on the same machine competing for the same resources is unacceptable.

9 Conclusion

David is born out of the frustration in doing large-scale experimentation on realistic storage hardware – a problem many in the storage community face. David makes it practical to experiment with benchmarks that were otherwise infeasible to run on a given system, by transparently scaling down the storage capacity required to run the workload. The available backing store under David can be orders of magnitude smaller than the target device.

David ensures accuracy of benchmarking results by using a detailed storage model to predict the runtime. In the future, we plan to extend David to include support for a number of other useful storage devices and configurations. In particular, the Storage Model can be extended to support flash-based SSDs. We believe David will be a useful emulator for file and storage system evaluation.

References

[1] GraySort Benchmark. `http://sortbenchmark.org/FAQ.htm#gray`.

[2] Microsoft longitudinal study dataset. `http://iotta.snia.org/traces/list/Static`.

[3] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, Feb 2009.

[4] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.

[5] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[6] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[7] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate i/o subsystem designs. *IEEE Trans. Comput.*, 47(6):667–678, 1998.

[8] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger. Timing-accurate Storage Emulation. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[9] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time-warped network emulation. In *NSDI'06*. USENIX Association, 2006.

[10] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[11] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world ai systems. *Artificial Intelligence for Applications, Conference on*, 0:87, 1995.

[12] E. L. Miller. Towards scalable benchmarks for mass storage systems. In *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996.

[13] Richard McDougall. Filebench: Application level file system benchmark. `http://www.solarisinternals.com/si/tools/filebench/index.php`.

[14] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

[15] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI '04*, Dec. 2004.

[16] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[17] E. Shriver. *Performance modeling for realistic storage devices*. PhD thesis, New York, NY, USA, 1997.

[18] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[19] Standard Performance Evaluation Corporation. SPECmail2009 Benchmark. `http://www.spec.org/mail2009/`.

[20] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[21] A. Traeger and E. Zadok. How to cheat at benchmarking. In *USENIX FAST Birds of a feather session*, San Francisco, CA, Feb 2009.

[22] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[23] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[24] M. Wittle and B. E. Keith. LADDIS: The next generation in NFS file server benchmarking. In *USENIX Summer*, pages 111–128, 1993.

[25] E. Zadok. File and storage systems benchmarking workshop. UC Santa Cruz, CA, May 2008.