Towards Realistic File-System Benchmarks with CodeMRI

Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison
{nitina, dusseau, remzi}@cs.wisc.edu

Abstract

Benchmarks are crucial to understanding software systems and assessing their performance. In file-system research, synthetic benchmarks are accepted and widely used as substitutes for more realistic and complex workloads. However, synthetic benchmarks are largely based on the benchmark writer's interpretation of the real workload, and how it exercises the system API. This is insufficient since even a simple operation through the API may end up exercising the file system in very different ways due to effects of features such as caching and prefetching. In this paper, we describe our first steps in creating "realistic synthetic" benchmarks by building a tool, CodeMRI. CodeMRI leverages file-system domain knowledge and a small amount of system profiling in order to better understand how the benchmark is stressing the system and to deconstruct its workload.

1 Introduction

Everyone cares about data, from scientists running simulations to families storing photos and tax returns. Thus, the file and storage systems that store and retrieve our important data play an essential role in our computer systems. To handle the different needs of various user communities, many different file and storage systems have been developed, from the Google File System [11], IBM GPFS [22] and NetApp Data ONTAP storage system [10] to local file systems such as NTFS [26] and Linux ext3 [32].

Unfortunately, file and storage systems are currently difficult to benchmark [31]. There is little consensus regarding the workloads that matter and insufficient infrastructure to make it easy to run interesting workloads. To evaluate the performance of a file and storage system, developers have a few different options, each with its own set of disadvantages.

•**Real Applications:** One option for evaluating a file or storage system is to directly measure its performance when running real I/O-intensive applications. The obvious advantage of benchmarking with real applications is that the performance results can correspond to actual scenarios in which the system will be used and that users care about. However, the problem is that real I/O-intensive applications can be difficult to obtain, to setup, and to configure correctly [30]. Often system evaluators compromise by running "real applications" that they are the most familiar with, such as compiling an operating system kernel or untarring a source tree. While these workloads are easier to setup, they may not be fully representative of the end applications.

•**Microbenchmarks of Application Kernels:** A second option is to run application kernels instead of the full applications themselves. For example, instead of configuring and stressing a mail server, one can instead run the PostMark [14] benchmark, which attempts to produce the same file system traffic as a real mail server. Other examples of kernels include the Andrew Benchmark [13] and SPC-1,2 [27]. While these kernels are simpler to run, they have the fundamental problem that their simplifications both may make them no longer representative of the original workload and enable system designers to artificially optimize to specific kernels.

•**Trace replay:** A third option is to replay file system traces that have been previously gathered at various research and industrial sites. Examples of traces include file system traces from HP Labs [21] and a collection of I/O and file system traces available through SNIA's IOTTA Repository [25]. Replaying file system traces eliminates the need to setup and recreate the original applications, but has challenges of its own. In particular, replaying the trace while accurately preserving the original timing [3] and accounting for dependencies across I/O requests [16] are non-trivial problems. In addition, traces are often large, unwieldy, and difficult to use.

•**Synthetic workloads:** A final option is to run synthetic workloads that are designed to stress file systems appropriately, even as technologies change. Synthetic workloads, such as IOZone [19], SPECsfs97 [33], SynRGen [9], fstress [2], and Chen's self-scaling benchmark [7], contain a mix of POSIX file operations that can be relatively scaled to stress different aspects of the system. The major advantages of synthetic applications is how simple they are to run and that they can be adapted as desired. However, the major drawback of synthetic workloads is that they may not be representative of any real workloads that users care about.

We believe that the ideal benchmark for file and storage systems combines the *ease of use* of synthetic benchmarks with the *representativeness* of real workloads. Thus, the goal of this paper is to describe how one can create realistic synthetic benchmarks. Specifically, our approach is to provide a tool that enables one to create a synthetic benchmark that is functionally equivalent to a given real application; that is, the synthetic benchmark stresses the system in the same way as the original application.

Determining whether or not two workloads stress a system in the same way is a challenging question; certainly, the domain of the system under test has a large impact on which features of the two workloads must be identical for the resulting performance to be identical. For example, if the system under test is a hardware cache, then the two workloads might need to have identical addresses for all issued instructions and referenced data; on the other hand, if the system under test is a network protocol, the two workloads might need to have the same timing between requests to/from the same remote nodes. Therefore, the specific features of the real workload that must be captured by the synthetic benchmark depend on the system.

One might believe that an equivalent synthetic workload for file and storage systems could be created by simply mimicking the system calls through the file system API (*e.g.*, read, write, open, close, delete, mkdir, rmdir). Given that tools such as strace [28] already exist to collect system call traces, creating such a synthetic workload would be relatively straight-forward. The problem is that system calls that appear identical (*i.e.*, have the exact same parameters) can end up exercising the file system in very different ways and having radically different performance.

File systems are complex pieces of system code containing hundreds of thousands of lines of code spread across many modules and source files. Modern file systems contain code to perform caching, prefetching, journaling, storage allocation, and even failure handling; predicting which of these features will be employed by a given system call is not straight-forward. Furthermore, the storage devices that are physically storing the data have complex performance characteristics; accesses to sequential blocks have orders of magnitude better performance than accesses to random blocks.

Consider the example of a `read` operation issued through the API. This read might be serviced from the file-system buffer cache, it might be part of a sequential stream to the disk or a random stream, or could involve reading additional file system meta-data from the disk. Similarly, a `write` operation might allocate new space, overwrite existing data, update file-system metadata, or be buffered as part of a "delayed write". In each of these cases, the exercised code and the resulting performance will be significantly different.

Our hypothesis is that to create an equivalent synthetic benchmark for file and storage systems, one must mimic not the system calls, but the *function calls* exercised during workload execution, in order to be functionally equivalent. We believe that if two workloads execute roughly the same set of function calls within the file system, that they will be roughly equivalent to one another.

In this paper, we describe our first steps in this direction by building a tool, CodeMRI (an "MRI" for Code, if you will). CodeMRI uses detailed analysis of the source code for the system under test to understand how a workload is stressing it. Specifically, CodeMRI measures function-call invocation patterns and counts to identify internal system behavior. Our initial results in applying CodeMRI to macro-workloads and benchmarks such as PostMark [14] on the Linux ext3 [32] file system are promising.

First, CodeMRI is able to deconstruct complex workloads into micro-workloads; each micro-workload contains system calls (*e.g.*, read and write) with known internal behavior (*e.g.*, hitting in the file buffer cache or causing sequential versus random disk accesses). In other words, with good accuracy, we are able to identify that a "real" workload, such as PostMark, performs the same set of file system function calls as a combination of system calls with certain parameters. Second, we are able to predict the runtime of the workload based on this set of constituent micro-workloads. We are working to improve CodeMRI to deconstruct more real workloads and traces [21], in order to create their synthetic equivalents.

The rest of the paper is as follows: We discuss CodeMRI in more detail in Section 2, discuss some challenges in Section 3, present related work in Section 4, conclusions and future work in Section 5.

2 CodeMRI

2.1 Introduction

The goal of CodeMRI is to be able to construct synthetic equivalents of real workloads. But there are two challenges in solving this problem. First, we need to accurately deconstruct real workloads into constituent *micro-workloads*. A micro-workload is a simple, easy to understand workload such as the `read` system call in its many forms, each with the same behavior (cached or not, sequential or random). Second, we need to be able to use the set of micro-workloads to compose a synthetic equivalent of the original workload.

We plan to approach this problem by leveraging two sources of information. First, we leverage domain knowledge about the system under test. Second, we use tracing to obtain useful information about the workload execution and the system under test.

Domain knowledge about file systems consists of basic knowledge about the different features that it provides, such as caching and prefetching. This is useful to know because different workloads can exercise different system features that CodeMRI needs to identify. The domain knowledge guides the tracing of execution profiles for micro-workloads. For example, we need to have an execution profile for a *cached read*.

For tracing the workload execution, we believe function invocation patterns and invocation counts provide the amount of detail necessary to understand the benchmark workload and the functionality that it exercises. This constitutes the *execution profile* of the workload. Our technique relies on some amount of tracing using a statically instrumented version of the system under test.

In order to address the first challenge – to breakdown real workloads into simpler micro-workloads, we compare the execution profile of a real workload with the set of execution profiles of individual micro-workloads. We call the execution profile of a micro-workload a *microprofile*. To address the second challenge – to synthesize a synthetic equivalent, we intend to compose the microprofiles together, along with timing and ordering information. Microprofiles are thus the building blocks for achieving both our objectives.

2.2 Building Microprofiles

The first step in building CodeMRI is to identify a comprehensive set of micro-workloads and build their microprofiles. We achieve this by running all the system calls through the file system API, under the effect of various file-system features. For example, in the case of a `read` system call, we build microprofiles for read, cached read, sequential read, and random read.

Through our experiments we find that keeping track of sets of function invocations and their counts, during a workload execution, allows us to build accurate microprofiles. We also observe that it is cumbersome and unnecessary to keep the entire execution profile – instead we select a small set of function invocations that uniquely characterize a microprofile. We call this the *predictor set* of a microprofile, and consequently the corresponding micro-workload. A predictor set typically consists of one to few tens of function calls, depending on the number of micro-workloads. The intuition behind this approach is that each function contributes towards completion of a higher level workload such as a `read`. Each function thus serves as the smallest unit of "useful work". The goal is to identify a set of functions that uniquely represent the higher level workload. In order to identify the set of function calls that constitute the predictor set for a workload, from amongst all the possible functions that contribute, we define some metrics to help automate the task.

We have three quantitative metrics associated with a predictor set – *slope*, *uniqueness* and *stability*. Two of these, uniqueness and stability (both on a scale of 0 to 1), are used in the selection of predictor sets. Each member function in a predictor set has a *slope* which characterizes the rate of change of invocation count with change in some workload parameter (such as request size). We define the *uniqueness* of a predictor set towards a micro-workload (such as `read`) as its affinity with the micro-workload. A uniqueness of 1 implies that the particular predictor set is invoked exclusively during this workload's execution, while 0.5 implies that it has an equal affinity with another workload, and 0 makes it irrelevant for that workload. The *stability* of a predictor set is a measure of the variability of function-invocation counts as some workload parameter is varied. A perfectly stable predictor set (i.e., with stability equal to 1) will scale proportional to the *slope*, as the request size of the workload is increased, for instance. A stability of 0 means that the predictor set scales in a completely uncorrelated fashion and is useless for prediction. Thus, an ideal predictor set for a given workload is one having both uniqueness and stability equal to 1.

Figure 1 shows a simple example of the predictor set for sequential reads having two member functions with scaling *slopes* equal to 0.85 and 0.895, *uniqueness* of 1, and *stability* very close to 1. This makes it a good candidate for being a predictor set to identify sequential reads.

To compute the slope for member functions in a predictor set, we keep track of their invocation counts, as we vary a workload parameter. This is done for a small "training range" to create a model. For example, in Figure 1, the training range for the request size model is from 200 to 1200 file system blocks. We found that predictor sets identified by CodeMRI have excellent stability that extends well beyond the training-model range.

The predictor set allows us to accurately predict the extent of the corresponding micro-workload. For example, if we observe a call to the function `ext3_readpages` and `ext3_block_to_path` a certain number of times, we can infer the corresponding bytes of random read being performed. Similarly the predictor set for cached reads corresponds to the amount of bytes being serviced from the buffer cache during a `read` operation.

The choice of a predictor set for any workload is not constant. For a single micro-workload, it is easy to find a predictor set with uniqueness equal to 1. However, for a real, complex workload, consisting of potentially tens to hundreds of micro-workloads, there can be significant overlap in the set of function invocations amongst the different micro-workloads, such that finding a predictor set for each of them is not straightforward. The size of the predictor sets depends on the number of micro-workloads to be deconstructed from the real workload. The more complex the real workload, the greater the number of functions required to construct predictor sets for each of the micro-workload.

CodeMRI consists of an algorithm based on linear-programming (LP) to select predictor sets, attempting to maximize the uniqueness for each of the micro-workloads. The LP problem constraints are in the form of minimum acceptable values for slope and stability, wherein the predictor set consists of the top-K functions

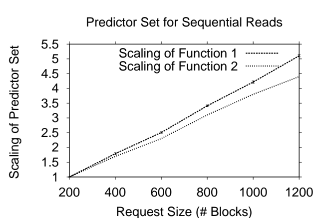

Figure 1: **Predictor Set.** *Having two member functions with scaling slopes equal to* 0.85 *and* 0.895*,* uniqueness *of* 1*, and* stability *very close to* 1

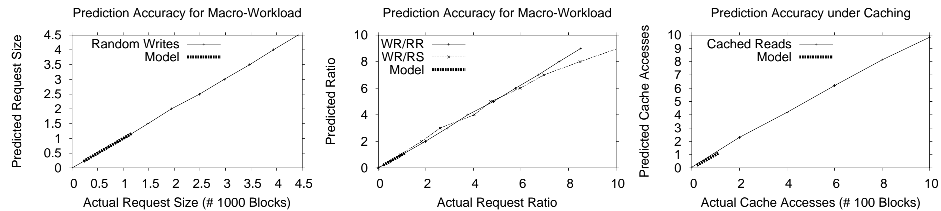

Figure 2: **Accuracy for Macro-Workloads.** *The leftmost graph shows the accurate prediction of random writes in a macro-workload consisting of writes and reads (both random and sequential). The middle graph shows the ratio of random writes (WR) to read sequential (RS) and random (RR) as predicted by CodeMRI. These ratios show that the relative counts across workloads are also accurate. The rightmost graph shows the shows the accuracy of prediction under caching. The Model line represents the training range on which the slope was computed.*

Micro-Workload	Variations	Parameter Extent or Count
Read	sequential or random cached or not cached	degree of randomness degree of caching
Write	sequential or random	degree of randomness
POSIX calls	open, mkdir, rmdir, create, delete, close cached calls e.g., open after create	count degree of caching

Table 1: **List of Micro-Workloads Tested.** *The table lists the various micro-workloads and their variations that were deconstructed with CodeMRI, along with the parameter of interest that was successfully predicted.*

that satisfy the given criteria, with K being the number of micro-workloads under consideration. In the absence of any function that satisfies the given slope and stability criteria, the conditions are relaxed until a match is found.

In practice, we find that most micro-workloads exhibit a natural division of function invocations, making it feasible to select high-quality predictor sets that give accurate results during workload deconstruction.

2.3 Using Microprofiles for Deconstruction

We now describe the use of microprofiles to deconstruct workloads. We present our discussion with increasing complexity of benchmark workloads:

- micro-workload, such as a `read` or `write`
- macro-workload consisting of micro-workloads
- macro-workload under caching
- application kernel: PostMark [14]

We find that CodeMRI has near-perfect accuracy for all the individual micro-workloads (not shown). Table 1 shows the list of micro-workloads that we have experimented with and are able to predict with good accuracy.

The accuracy continues to be good for macro-workloads. Figure 2 shows the accuracy of prediction for a macro-workload consisting of `reads`, `writes` and system calls such as `open` and `close`, as we vary the request size. The leftmost graph shows the accurate prediction of random writes, and the middle one shows the ratio of writes to sequential and random reads as predicted. The rightmost graph in Figure 2 shows the accuracy of prediction of reads under caching. In these graphs, the "Model" line represents the training range on which the slope for prediction was computed.

CodeMRI is thus able to accurately identify workloads well beyond the small training range of request sizes for which the slope model was computed. Only for larger deviations from this range do we observe inaccuracy.

In order to verify whether this deconstructed workload has any correlation with actual performance, we use it to predict performance and compare with the actual measured performance. The hypothesis is that if the deconstruction is accurate, then the sum of time taken by the individual micro-workloads should be close to the actual measured time. To predict performance once we have identified the set of micro-workloads, we simply add the time it takes to run them individually. This is a coarse estimate, as it does not take into account dependencies amongst the micro-workloads.

Figure 3 shows an example of this for a macro-workload consisting of random and sequential reads, `mkdir`, `create`, and `delete` operations. The left graph highlights that the primary contributor(s) to performance can be different from the expected ones, and CodeMRI can identify the real contributors. The "issued operations" are the ones issued through the file system API. The "actual operations" are the ones being actually issued by the file system to the disk, and not serviced from cache. The "predicted operations" are the ones identified through CodeMRI. In this example, random reads contribute much less to overall runtime than sequential reads and mkdir. The stacked bar graph on the right shows the predicted runtime contributions from individual micro-workloads. We see that the predicted cumulative runtime matches closely with the measured runtime, demonstrating the accuracy of CodeMRI.

CodeMRI thus not only deconstructs workloads accurately, but the deconstructed workload is useful in predicting performance. We find that the actual runtime of the workload is in accordance with the predicted workload.

We next deconstruct a popular file-system benchmark, PostMark [14], that simulates an email workload, and find

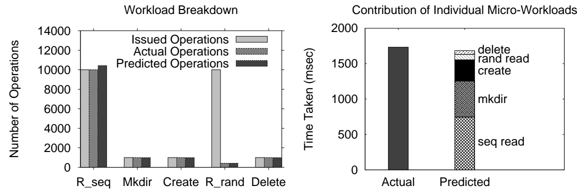

Figure 3: **Macro-workload Deconstruction.** *The macro-benchmark consists of* `mkdir`, `create`, `delete`, *repeated random reads to a small file, and sequential reads to a large file, resulting in random reads hitting the cache. The graph on the left shows the deconstruction of this macro-workload by CMRI which identifies the effective (reduced) count of random reads. The graph on the right shows the individual contribution of different micro-workloads towards the total runtime, as predicted by CodeMRI.*

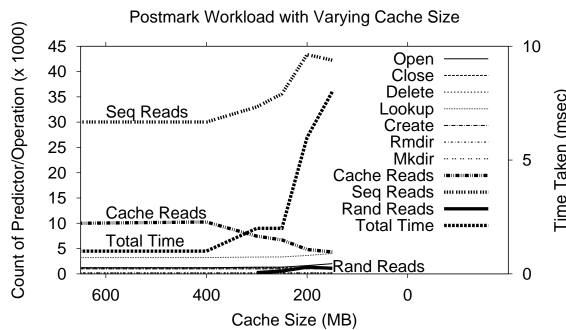

Figure 4: **PostMark with Decreasing Cache Size.** *The postmark configuration is 1000 files, 200 subdirectories, 4K block size, 1000 transactions, with other parameters as default. CodeMRI accurately deconstructs the effective workload and the total runtime is in accordance with the predicted workload. Important micro-workloads are in thicker lines.*

that the accuracy is near-perfect (not shown), as in the previous cases. We investigate the PostMark workload one step further. Figure 4 shows the breakdown of PostMark's workload under varying cache sizes. As the size of cache decreases, CodeMRI is able to identify its effect on the workload, as there are fewer cached reads, and sequential reads becoming increasingly "random". The total runtime of the benchmark is also proportional to the overall effective workload, making it a useful performance predictor.

Our evaluation of CodeMRI's accuracy for a number of simple to more complex workloads gives us confidence in applying it on real applications and file-system traces. We are trying to incrementally increase the complexity of the input workload, and are currently deconstructing file-system traces, such as one from HP Labs [21], with the final goal of running CodeMRI for real applications.

2.4 Using Microprofiles for Synthesis

The final step in the construction of realistic synthetic benchmarks is to use the microprofiles to synthesize an equivalent workload. In addition to building the microprofiles, we also need to maintain timing and ordering information, in order to issue the micro-workloads while accurately preserving the original timing, and accounting for dependencies across I/O requests. Our current implementation has the necessary setup to collect most of this information. We continue to refine CodeMRI in order to synthesize benchmarks equivalent to real workloads.

3 Challenges

In this section we discuss potential challenges in building CodeMRI, and some alternate approaches.

Another approach would be to use tools such as strace [28] to collect system calls for real applications and replay the trace. This alone will not be useful, since similar calls through the API can end up exercising the file system in very different ways and have radically different performance due to effects of caching and prefetching.

A more effective solution will be to obtain both system call and disk traces to account for file system policies and mechanisms. But correlating strace and disk trace information is not entirely straightforward due to timing issues, especially in presence of buffering and journaling. Furthermore, the disk I/O might be reordered or delayed, and be affected by file system daemons such as `pdflush`.

Semantic Block Analysis [20] is another means to infer file system level behavior, but requires detailed file system knowledge. CodeMRI has the added advantage of being oblivious of the file system in question.

In its current form, CodeMRI needs source code for analysis, which can somewhat limit its scope. However, there is nothing fundamentally limiting CodeMRI to require instrumented source code. In order to collect the execution profile of a workload, tools such as Kerninst [29] can be used. They have the advantage of tracing unmodified binaries, without requiring source code access.

Factors such as configuration parameters, hardware settings and real-time traffic can also affect the performance of a system. These factors cannot be captured by the application source code alone. Since CodeMRI is designed to operate on the running workload (and not as a static analysis), it is going to capture the effects of these external factors as they drive the source code into different regimes of operation. The goal of CodeMRI is to deconstruct a running workload, as opposed to generating all the different workload execution scenarios.

4 Related Work

We leverage system profiling and file system domain knowledge to understand internal system behavior during execution of real workloads, and use that to create synthetic equivalents of the workload. Several tools already exist for instrumenting and profiling systems, such

as, Kerninst [29], Dtrace [5] and gprof [12]. For our analysis, the level of instrumentation provided by static instrumentation was sufficient since almost all of CodeMRI's logic lies outside of tracing. In the future, more sophisticated profiling tools can be integrated. Similar to our work, performance debugging of complex distributed systems [1, 4, 6] also uses tracing at various points to infer causal paths, diagnose and tune performance bottlenecks, and even to detect failures using runtime path analysis. A number of tools have been developed to understand, deconstruct and debug complex software systems such as Simpoint [24] and Shear [8]. Delta debugging is another technique that uses an automated testing framework to compare program runs and access the state of an executable program to prove the causes of program failures [34]. Mesnier *et al.* have proposed "relative fitness" models for predicting performance differences between a pair of storage devices [17]. A relative model captures the workload-device feedback, and the performance and utilization of one device can be used in predicting the performance of another device. This shifts the problem from identifying workload characteristics to device characteristics. In the future, it will be interesting to use CodeMRI together with relative fitness models. Finally, we benefited from opinions expressed in other "Hot" papers on benchmarking [18, 23].

5 Conclusions and Future Work

We have presented our first steps in building CodeMRI – a tool that enables the construction of realistic synthetic benchmarks from real workloads and file-system traces. Our initial results in applying CodeMRI to simple workloads have been promising. We intend to continue improving its accuracy for more real-world workloads.

Several challenges remain to be addressed – our current implementation is meant to illustrate the benefits of CodeMRI and is not optimized for production environments. In practice, we find that the small amount of tracing doesn't slow down the system appreciably, but optimizations for performance and accuracy are certainly possible. We also plan to explore use of statistical techniques similar to ones used in bug isolation [15] to improve accuracy and stability of predictions. Another goal is to minimize runtime variability due to concurrent activity and non-reproducible events (e.g., interrupts).

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. M. oen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03*.

[2] D. Anderson and J. Chase. Fstress: A flexible network file service benchmark. In *TR, Duke University, May 2002.*

[3] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *FAST '04*, San Francisco, CA, April 2004.

[4] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Real-Time Modeling and Performance-Aware Systems. In *HotOS '03*.

[5] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, pages 15–28.

[6] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *NSDI '04*, San Francisco, CA, March 2004.

[7] 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 *SIGMETRICS '93*.

[8] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *ASPLOS XI*, pages 59–71, Boston, MA, October 2004.

[9] M. R. Ebling and M. Satyanarayanan. Synrgen: an extensible file reference generator. In *SIGMETRICS '94*.

[10] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and C. Wagner. Data ontap gx: a scalable storage cluster. In *FAST'07*.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP '03*.

[12] Gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html. 1998.

[13] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM TOCS*, February 1988.

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

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05*.

[16] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. trace: parallel trace replay with approximate causal events. In *FAST '07*.

[17] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage.

[18] J. C. Mogul. Brittle metrics in operating systems research. In *HotOS '99*.

[19] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[20] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *USENIX '05*, pages 105–120, Anaheim, CA, April 2005.

[21] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*.

[22] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02*.

[23] M. I. Seltzer, D. Krinsky, K. A. Smith, and X. Zhang. The case for application-specific benchmarking. In *HotOS*, 1999.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS '02*.

[25] SNIA. Storage network industry association: Iotta repository. http://iotta.snia.org, 2007.

[26] D. A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[27] SPC. Storage performance council. http://www.storageperformance.org/, 2007.

[28] strace. http://linux.die.net/man/1/strace. 2008.

[29] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99*.

[30] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and Batch Sharing in Grid Workloads. In *HPDC 12*, pages 152–161, Seattle, WA, June 2003.

[31] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. Accepted for publication, ETA February 2008.

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

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

[34] A. Zeller. Isolating cause-effect chains from computer programs. In *10th FSE*, 2002.