Bertha: A Benchmark Tool for High-Performance Storage Subsystems

Dave Wagoner
wagdalule@hotmail.com

The popular circa 1990 UNIX file system benchmark tool "Bonnie" has been substantially modified to form a distinct benchmark tool called "Bertha". The features of Bertha that distinguish it from Bonnie, as well as most other benchmark tools, are 1) it generates substantially more I/O to challenge contemporary storage subsystems so that their capabilities can be better measured, 2) it offers a unique facility to "replay" I/O "transactions" that have been captured from actual production applications or simulations to more accurately suggest performance that will be experienced when using particular applications, and 3) extensive metrics reporting that provides more insight than what a single average value can provide.

The Bonnie benchmark tool, authored by Tim Bray, was a popular tool used in the 1990s to measure UNIX file system performance. The load generated by Bonnie stressed operating systems and storage subsystems from the perspective of an application, thus its measurements were afflicted with the distortions caused by various caches. For a sufficiently large load, the cache could be over-ridden and the impact of the cache minimized, but not completely negated.

At the time Bonnie was written, a single CPU in most system configurations was generally sufficient to generate enough I/O load to saturate a disk subsystem. By the late 1990s, this was no longer the case – running Bonnie could consume an entire CPU without reaching the maximum throughput of a disk subsystem. Under these conditions, the results from a benchmark would be the same regardless of the storage subsystem configuration as long as it met the minimum workload threshold to saturate a CPU. Identifying the impact of tuning changes, comparing storage products, and stress-testing systems could no longer reasonably be done with Bonnie.

What made Bonnie attractive was that it was an elegant tool written in a few hundred lines of code that was easily deciphered and modified. With a few modifications, the core Bonnie code was modified to generate substantially more load. The methodology for doing this is to spawn a set of processes to generate I/O operations (IOPs) concurrently in a coordinated fashion. Thus, rather than a single thread of execution generating sequential IOPS, an arbitrary number of processes can be made to generate many times more IOPs to saturate file systems and storage subsystems and measure their true upper-bound performance.

In considering benchmarks, the conventional wisdom is that the only benchmark that matters is one's own application. With this in mind while enhancing Bonnie, it became evident that a facility to play back or "replay" captured IOPs from actual production applications could be implemented with relative ease. In addition to replaying application transactions, simulations for "what-if" types of analysis could also be facilitated. The content of the IOPs from the actual application are not replayed. The replayed "transactions" preserve the I/O access pattern, the specific transaction (read or write, operations), and the data volume. Buffers for write operations were altered ("dirtied") before each write operation so as to attempt to thwart I/O avoidance mechanisms.

The original Bonnie benchmark tool ran six basic tests: *write*, *read*, *rewrite*, *character-based read*, *character-base write*, and *random*. The *write* test created a scratch (I/O test) file; the read test simply reads the data from the scratch file created during the *write* test. The *re-write* test consisted of reading a buffer of I/O from the file, moving back to the beginning of where the read data started (i.e. seek back to the beginning of the data just read), then re-writing the data, moving to the next sequential data and repeating until the entire file has been re-written. The *character-based read*, *character-based write*, and *random* tests focused more on file system performance than on storage subsystem performance. Because memory has become cheaper and more plentiful than it was in the 1990s, these particular tests have perhaps diminishing value as caching dramatically aids in I/O avoidance. Reported throughput rates for these tests became unrealistically high and it was obvious that I/O was in most cases not truly being performed. These tests were consequently removed, leaving *write*, *read*, and *rewrite* (and *replay*) tests in the Bertha implementation.

Reconsideration may be in order if evidence emerges that the removed tests can be effectively used to measure performance of storage subsystems and not just cache performance.

The two enhancements of the capability of increasing I/O load and the capability of performing I/O transaction playback (a.k.a. replay), particularly the former, were used for a number of years in limited settings in the late 1990s and early 2000s by the author and co-workers. The reported metrics were still averages of IOP throughput as reported by Bonnie – a single number to describe the throughput of each test. Averages can often lead to a distorted view of results, particularly when outliers are substantial.

Further, the enhancement of using multiple concurrent processes to generate I/O greatly magnifies the unsuitability of a single average value to characterize performance. Although processes that are assigned to do a fixed amount of I/O can be synchronized at the start of a test, the processes do not terminate at the same time; some processes continue to generate I/O whilst others have long since completed. Depending on the parameters selected for a particular benchmark, the time during which some number less than the command line-specified number of processes are running, particularly near the end of the test, to generate I/O could be substantial.

Note that a distinction is being made here between "benchmarks" and "benchmark tools." Despite the popular use of the terms, Bonnie and Bertha should be referenced as "benchmark tools" that perform some user specified test. A benchmark would consist of the benchmark tool and a specific set of supplied parameters, such as the number of concurrent processes to run, the I/O size for each operation, and the size of the scratch files on which the IOPs are to be performed.

To address the issue of having a perhaps nebulous single number to describe the outcome of a test, a scheme was devised to record the timestamp of initiation and response time of every I/O transaction performed by Bertha. This enables a much more detailed view of what occurred during a test to be provided, yielding insights into storage subsystem behavior and exposing characteristics of particular benchmarks warranting attention. Effects of storage subsystem cache, for example, become clearly evident as bursts of IOPs are performed at the beginning of a *write* test, followed by long periods of inactivity, followed again by a burst of IOPs in a repeating pattern. It is worth noting that Bertha is reporting response times that are perceived by applications; what may be occurring at the operating system level may be somewhat different. However, users are generally most interested in performance at the application level. Performance at lower levels (operating systems, storage subsystems) is perhaps of academic interest, but secondary to what is experienced by applications.

Having this level of detail allows a wealth of various reports of interest to be generated. In particular, a timeline can be constructed, showing IOP volumes and IOP concurrency levels (the number of IOPs being performed at any given time). Histograms of response times, data throughputs, and IOP concurrency levels are also generated. Summary statistics showing median, minimum, maximum, and various percentile response times are generated to provide a high level description of the results. Last, the availability of this level of detail allows response time vs. throughput and response time vs. IOP concurrency level summaries to be derived. Obviously, other types of reports can be generated; the ones listed here were the ones that initially appeared useful.

In addition to being a tool for measuring performance, another use for Bertha is testing of the resiliency of configurations during the failure of selected components. For example, measuring the impact of the removal of one of two fiber cables that are carrying load-balanced IOPs can be easily facilitated. The time of the cable pull can be easily matched with the timeline reports generated from the test to determine if an impact on performance, a stall in activity, or some other effect resulted.

Throughput Testing

As mentioned earlier, Bertha will measure throughput for three basic operations: *write*, *read*, and *rewrite*. IOPs for each of these tests are performed by a set of `fork`(2)ed processes that start operating at approximately[1] the same time. The number of these concurrent processes is specified on the command line when Bertha is invoked.

Each test is performed to completion before the next starts. All processes performing a particular test (*write*, *read*, or *rewrite*) must be completed and I/O activity quiesced before the next test begins. A `sync`(2) is executed between each step, and a delay of several seconds exists between the tests to minimize the residual impacts of each test on the next test. The processes performing a particular test terminate upon completion of a test and new processes are `fork`(2)ed at the beginning of the next test.

A distinction in terminology is made here between *concurrent* and *parallel*. *Parallel* implies that "activities"

[1] All processed are typically `fork`(2)ed in less than one second.

that are said to be parallel start at the same time and stop at the same time. *Concurrent* is a somewhat looser constraint where activities overlap in time; that is, activities are said to be concurrent if one starts before another has completed or if one continues whilst another has completed. Bertha processes performing IOPS for throughput tests start at approximately the same time but end at potentially widely varying times, hence the looser term *concurrent* is used rather than *parallel*.

Each process performs its I/O to an exclusively accessed file referenced as a *scratch* file[2]. This file is created during the *write* test and used in subsequent tests. At the completion of a Bertha run, these scratch files are deleted. The length of this file is provided as a command line argument in units of megabytes. Each process performs its particular test on the entire scratch file before completing; that is, the length of the Bertha run is the amount of time necessary to perform the tests on a fixed amount of I/O that is user specified at the time of execution.

In addition to supplying the number of processes and scratch file size, the size of each IOP is also supplied in units of bytes; the default is 16384 (16 Kbytes). Each read and write operation will be of this length from an application level, but the operating system may combine application level IOPs together and perform fewer but larger IOPs as an optimization. This was particularly evident in testing on Solaris 9, where the system activity reporter facility, `sar`(1), showed throughput volumes equal to what Bertha reported at an application level, but `sar`(1) showed a reduction in number of IOPs performed by an order of magnitude when using an I/O size of 16 Kbytes.

Using threads rather than `fork`(2)ed processes to increase I/O concurrency for higher throughput levels is an alternate implementation. However, one of the design objectives was to make Bertha as portable as possible, relying on "lowest common denominator" interfaces, libraries, and mechanisms to increase the likelihood of successful porting to a variety of platforms[3]. Threading implementations can vary dramatically between operating systems, such as Redhat Enterprise Linux (RHEL) 3 and RHEL 4, but the use of `fork`(2) to spawn processes should provide more consistent behavior. In retrospect, the use of threading, depending on the operating system implementation of the threading model, may remove the need to have shared memory segments (described in the section below). The advantages of not having shared memory segments are minimal, however, and it is somewhat debatable as to whether other advantages exist.

Replay Capability

In addition to the throughput tests that have a set of processes that each perform sequential I/O, the replay capability implemented in Bertha allows a set of specific I/O transactions to be executed to observe performance characteristics that may be expected by an application on the particular configuration being tested. These transactions can be captured from actual applications running in a production environment, constructed programmatically or manually with an editor to implement a simulation, or a combination of the two (described in more detail below).

The I/O transactions are stored in an ASCII file referenced as a "trace" file. The first line of the trace file contains the length in bytes of the file being accessed by the real or simulated application; all other records contain the I/O transactions, one per line. Each I/O transaction is composed of a starting location, an operation (read or write specified by the single characters 'r' or 'w', respectively), the length of data in bytes to read or write, and the length of time in seconds to wait after the transaction before performing the next I/O transaction. The content of data being written will not be what the application might see, but rather the data will be a "dirtied" buffer of the size specified for the particular transaction. The locations addressed in the I/O transactions of the trace file are all confined between the address of 0 and the file length specified on the first line of the trace file. Attempts to access beyond these bounds result in an error and termination of a Bertha run.

The delay component of the replay I/O transactions is analogous to "think time" used in interactive applications. To determine maximum throughput, delay values will be zero. The specification is in units of seconds, but not confined to integer values. The library call `usleep`(3) allows delay in the resolution to be specified in units of microseconds[4].

By default, each process run during a Bertha *replay* test, specified on the command line with the `--num_procs` command line argument, will perform the set of transactions specified in the trace file on its own

[2] The exception is replay mode with the `--replay_mono_file` is used; in that case, all processes are performing IOPs on the same file.

[3] At the time of writing, Bertha runs on Linux (RHEL 3 and RHEL 4), and Solaris 9.

[4] The library function `usleep`(3) has been noted as obsolete in its man page. Future versions of Bertha will need to use `nanosleep`(2) or `setitimer`(2) instead of `usleep`(3).

scratch file. Alternate semantics can be implemented with the --replay_mono_file flag, where each of the processes performs some approximately equal number of transactions specified in the trace file on a single file; that is, all processes are reading and writing to the same file. If the number of transactions in the trace file is not a multiple of the number of processes performing I/O, the nth process performs its allocated set of transactions from the trace file plus the "remaining" transactions.

The size of the file described in the trace file may not necessarily match the size of the scratch file as specified on the command line invoking Bertha. It is quite possible that the resources available in a test environment might not be equal to what is available in a production environment. Pursuing "what if" analyses would also be a reason why the file sizes may not match.

To address this situation, I/O addresses are proportionally scaled to map transactions to address the same relative locations in the files. That is, a mapping function uses a scaling factor from the file size specified in the trace file to the scratch file size. By default, the target addresses for I/O operations are scaled but the volume of data (the number of bytes read or written) is not. Using the --replay_scale_io_size command line flag causes the number of bytes in an I/O operation to be scaled in addition to the scaling of the target I/O address.

Actual production transactions can be captured in a number of ways. One way is to instrument applications to reveal I/O transactions. However, third party applications do not typically provide such a luxury. When such instrumentation is not available, it can be obtained by vxtrace when Veritas Volume Manager is being used. Further, truss(1) on Solaris or strace(1) on Linux implementations can be used to show all system calls being executed by a process, including read(2), write(2), and lseek(2) along with their arguments and return values. This data can be parsed and reformatted to form the I/O transactions to be placed into a trace file for replay. When using truss(1) or strace(1) output, the parsing program will need to trace the current location being addressed within the file. The vxtrace utility provides this data. Note that truss(1) and strace(1) show activity from an application perspective; vxtrace shows activity from an operating system perspective. Specifics on the format of the trace file records are described in the Bertha man page.

Metrics Reporting

Bonnie reports throughput rates (and CPU utilization) observed during each of the tests it runs. The values reported are averages for the duration of the test. When testing using a single process, as Bonnie does, this provides a quite acceptable characterization of I/O performance from an application perspective. However, when using multiple processes that may not terminate at the same time, use of a single average is of questionable validity.

To address this issue, all I/O transactions during Bertha runs are stored to more precisely characterize performance. Preserving the start time, response time, and I/O size allows statistical summarizations such as median, 90th percentile, standard deviation, maximum value, and others are available in addition to averages. Further, having this data means that a wealth of different reports can be used to investigate I/O subsystem behavior in more detail.

For example, the amount of data written over time during a *write* test reveals a repeating pattern of bursts of I/O followed by several seconds of delay. This is likely explained by buffering occurring at the operating system level and caching in the file system cache and the storage controller cache, as well as cache on DASD. Note that Bertha is a benchmark tool – it reveals particular behavior, it does not provide explanations as to why the behavior is there. In this sense, Bertha is analogous to a microscope in providing data.

In addition to activity over time, a set of histograms that characterize response time, throughput, and I/O concurrency level are available. The latter of these is the number of I/O operations active at a given time. The aggregation of data for the histograms is controlled by command line arguments that specify the time increments into which the data is gathered and the "bucket" size used for the aggregation for the particular measure (e.g. throughput and I/O concurrency).

Data is also aggregated to provide a basis for response time vs. throughput and response time vs. I/O concurrency level curves. These are categorized along with the histograms for ease of implementation.

Reports

Bertha currently provides reports in column-formatted and labeled text (ASCII) files or in comma separated value (CSV) files. There are plans to implement files that can be read by gnuplot, R, and SAS. Some or all of these additional report formats may be available by the time of publishing.

Command line arguments are used to specify which type(s) of reports are to be generated. Multiple report types can be generated for each run. The Bertha man page contains the specifics regarding what flags are used to cause the various reports to be created. Recall

that reports will be created in the directory specified by the --report_dir command line argument or default to "./reports" relative to the current working directory in use when Bertha is invoked.

"Re-reporting" Capability

Benchmarks conducted using Bertha may take a considerable amount of time. Upon reviewing results, it may be necessary to alter the histogram parameters to aggregate the results differently to produce more useful results summaries. Performing the benchmark again to obtain these can be quite costly with respect to time.

The "re-reporting" capability was added to Bertha to address this issue. This capability allows the raw results from a particular run to be stored in a data file and read again for reporting using different histogram parameters. When running to generate data, using the --record command line flag will cause data files for each test performed (*write*, *read*, *rewrite*, *replay*) to be stored in the directory specified to contain the reports. Using the --rereport command line flag will cause Bertha to look for data files corresponding to the tests performed and generate a new set of reports based on the histogram parameters supplied. No I/O testing is done when the --rereport command line flag has been specified.

Coding Details

The Bertha benchmark tool is written in the C programming language and is composed of approximately 6,000 lines of code. Considerable effort was made to produce high-quality, readable code. Whether this goal was achieved is as yet to be judged. It is anticipated that since this tool will be made available as an open source software package, others will peruse the code to identify oversights and implement enhancements; a focus on ease of readability and modification were therefore necessary.

The static source code checking tool splint was used on all Bertha source code. It is typical for splint to have numerous "false positives" with regard to perceived errors in source code, and running it on Bertha proved to be no exception. It is believed that all genuine issues identified by splint have been addressed.

A runtime tool named valgrind was used to verify the operation of Bertha with regard to the use of dynamically allocated memory. Bertha extensively uses dynamically allocated memory to contain metrics results and intermediate values when generating reports. valgrind showed that no memory leaks existed – all allocated memory was de-allocated prior to program termination.

Each C function within Bertha performs at least rudimentary pre-condition checks to ensure that function arguments are within expected range. While this may seem redundant, the expectation is that Bertha will be examined and modified by other open source contributors; the use of pre-condition checks will greatly aid in error detection efforts as new and modified code are introduced. In addition to pre-condition checks, rudimentary post-condition checks are used where appropriate.

The use of the assert(3) function is perhaps out of vogue, but it is relied upon extensively in Bertha. As with the apparently redundant pre-condition checks, the assert(3) invocations are left in the anticipation that the code will be modified and these invocations will greatly aid in problem identification.

Extensive use is also made of the standard global variable errno in Bertha code. It was once said in jest that Ken Thompson, one of the original authors of the first UNIX implementation, designed a car. As was the perceived case with UNIX, Thompson's fictitious car had no gauges, no dials, and no indicators – save for one large red light that resided in the center of the dashboard. "When it goes off," Thompson was to have said, "the user will know what the problem is." The errno variable is in a sense this red light, but considerably more useful.

The value of errno is zero when no error has occurred; once an issue with a system or library call has occurred, errno will contain some non-zero value that is specific to the system or library call invoked. Numerous uses of

```
assert(!errno)
```

are embedded where appropriate throughout the Bertha source code. These aided greatly in accelerating the development of Bertha. After the first version of Bertha was completed, the decision was made to leave these particular assert(3) invocations in the code. As with pre-conditions and other assert(3) invocations, it is expected that they will be instrumental in identifying issues as code is added or modified in Bertha by other open source contributors. The typical Bertha user who is not utilizing its source code should be oblivious to the existence of these invocations.

Some of the criticisms of assert have been focused on the additional execution time and larger program size. The sections of code that issue I/O instructions within loops are particularly time-sensitive; unnecessary code may degrade the quality of results, particularly when measuring maximum throughput. These particular sections of code are compact and have as little code in

them as possible to minimize the impact of the benchmark tool on the results.

Criticisms of Bertha

Few software packages pass without criticism. Included here are some of the current and likely criticisms of Bertha and relevant commentary. The current version of Bertha that is open source is version 1.0, but Bertha has been evolving over the last seven years. Further evolution will be guided by feedback from actual users and changes in available operating systems and storage technologies.

The most prominent criticism of Bertha from current users is that Bertha tests run for a substantial period of time. This must be the case because contemporary components used for testing, both host systems and storage subsystems, have large file system and I/O caches. To realize what will happen during production-level potential stress, the effects of caches must be negated by moving enough data to fill the caches several times.

It is particularly noteworthy that during Bertha tests, even when running as a non-privileged user, host systems have been put into a "hung" state and file systems containing the scratch files were rendered corrupt. This has not happened when testing "vanilla" (standard) operating system and file system configurations using UFS or VXFS. This has happened with alarming consistency when testing clustering software and global file system implementations, however. The length of time observed to be necessary to "hang" a host varies from one minute to approximately 40 minutes, depending on the specific software being tested.

The speculation based on observation and interaction with vendors is that in most cases, I/O queues handled in kernel structures are not being properly managed and non-I/O related data structures may be inadvertently overwritten. This is, however, speculation at this point as vendors have not been forthcoming with the specifics regarding why this behavior occurs, but have given guarded and vague responses. While not every global file system implementation has been tested, every one that has been tested with Bertha has suffered catastrophic failure. Bertha performance tests become functionality tests; this is actually a "blessing in disguise" and has reliably demonstrated flaws in the author's computing environment that were being exhibited on an intermittent basis at best. This aided in diagnoses and corrective actions, thus definitively resolving production environment problems.

A second criticism of Bertha will be related to the numerous report files that may be generated. These can be combined or used selectively. Input from users as to how to address this is invited.

Questions regarding the validity of Bertha tests with regard to reflecting actual production workloads often emerge. The answer is, as is often the case, "it depends." With regard to measuring maximum throughput of an I/O configuration, Bertha testing is likely to be exceptionally valid. Stress testing with throughput tests may well simulate backup or data loading operations that, while they may not be representative at present, data growth will likely make them valid in the future. The upper-bound of performance and assurance of functionality under sustained, heavy load (a.k.a. an endurance test), are needed when businesses are risking revenue on system performance and stability.

Further, the argument can be made that Bertha *replay* tests, which perform the I/O access pattern and data volumes for real or simulated applications, are extremely valid. The `--replay_scale_io_size` and `--replay_mono_file` options allow a variety of behaviors to be replicated to address needs. Other variant needs may emerge and be reported by users.

Examples

A simple example of a throughput test is shown below. This test was run on a development workstation with internal SCSI drives.

The command used to perform the test is shown below. The man page for bertha contains details on the options provided.

This particular test "sprayed" I/O at two distinct directories. This is dictated by the occurrence of multiple `--scratch_dir` arguments being supplied. This is useful where the directories would be mounted on distinct storage so that the goal of a particular test would not necessarily be to determine the maximum throughput of a particular LUN, but of an I/O channel or some other potentially constrained component.

Also noteworthy is that 10 concurrent processes are being used to generate I/O, as indicated by the `--num_procs` arguments. Each of these processes writes to its own scratch file of 100 Mbytes.

```
./bertha \
   --scratch_dir /fs/scratch_0 \
   --scratch_dir /fs/scratch_1 \
   --report_dir ./reports_test_2b \
   --num_procs 10 \
   --scratch_file_size 100 \
   --verbose --verbose --verbose\
   --logfile bertha_test_2b.log \
   --metrics_by_proc \
   --response_time_hist=[1.0,,1]        \
   --xput_hist=[0.1,5.0,1] \
   --concurrency_hist=[0.1,1.0,1] \
   --rsp_vs_xput=[1.0,0.5,1] \
```

```
   --rsp_vs_concurrency=[1.0,0.25,1] \
   --time_line=[0.1,,1] \
   --test_name test_2 \
   --sas_reports \
   --record \
2>&1
```

Bertha command used to perform throughput test

Rather than a single number showing average performance, a set of descriptive statistics is provided that summarizes all 64,000 write operations performed for this test. Note that the reported average is dramatically influenced by the maximum observed response time. This high value was likely generated when some component was saturated. The reported times are in units of milliseconds.

```
                Write Response Time

         Test Run: Sat Mar 25 23:52:58 2006

 Response Time (msec) Metrics Summary for Test Run

Min:          0.135183
Median:       0.299931
Avg:          3.324013
75 %tile:     0.384092
90 %tile:     0.516891
95 %tile:     0.599861
99 %tile:     0.996113
Max:       2860.401855
Num Vals:        64000
Stddev:   15162.135742
```

Summary statistics for response time

One of the types of reports generated is the data necessary to produce a histogram. Below is the output generated for a histogram of throughput during the *write* component of test.

```
                Histogram Table
         Write - Aggregate Throughput (MB/sec)

         Test Run: Sat Mar 25 23:52:58 2006

Intvl              Range                      Freq
           From    -    Up To

    3    15.000000 -    20.000000              1
    5    25.000000 -    30.000000              2
    6    30.000000 -    35.000000              1
    7    35.000000 -    40.000000              3
    8    40.000000 -    45.000000             16
    9    45.000000 -    50.000000              2

                                             ------
Total number of samples:                         25

Description: This histogram shows the number of
time intervals that were categorized into buckets
of size 5.000000 MB/sec. The frequency represents
the number of sample intervals where the aggregate
system I/O was in the range given by the bucket
size.
```

Note that the bucket sizes of the sample times and throughput are specified on the command line. The `--put_hist` parameter is the specific parameter set that affects this particular report. Note that the `--time_line` parameters were used for the graphs and have a different granularity for time than the parameters used for the throughput histograms. The ability to re-aggregate the data differently for subsequent reports produced during the same run provides considerable flexibility.

Below are the corresponding graphs for the write component of the test. Figure 1 shows throughput as observed every 0.1 seconds, as specified by the `--time_line` parameters. Figure 2 shows the I/O concurrency – the number of I/O operations active at any one time, also as observed every 0.1 seconds. Recall that 10 concurrent processes were being used to generate I/O.

The two graphs below were produced by SAS. By using the `--sas_reports` option, Bertha emitted the SAS code to load the data points into a SAS dataset and generate graphs. The user need only run Bertha, load the SAS code produced and run it to visualize the data. The user can then iteratively refine the histogram parameters and re-run Bertha with the `--rereports` option to refine the data presentation without having to generate I/O. This ability provides users with considerable flexibility in presenting results.

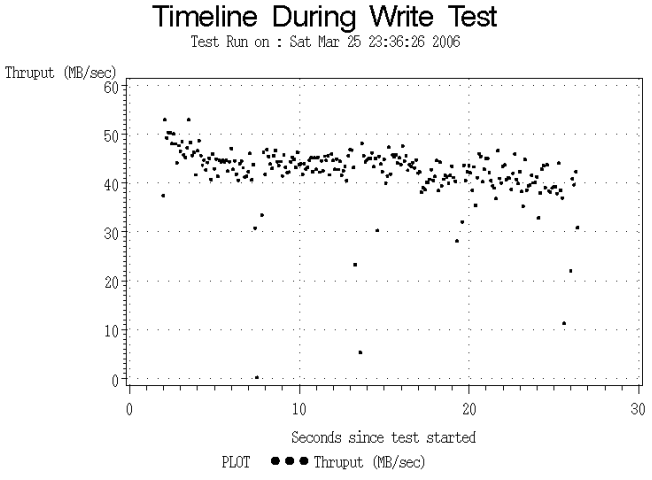

Figure 1: SAS graph of throughput during *write* test

Histograms are also generated. The write throughput histogram, again produced by SAS running bertha-generated code is shown below in Figure 3.

Bertha can also generate scripts and data files for the generation of charts with gnuplot and R as well as SAS. These files are generated by supplying the command line options `--gnuplot_reports` and `--R_reports`, respectively. The scripts and data files

will be located in the reports directory specified by the `--reports_dir` option or the default `./reports` subdirectory. Throughput results obtained during a non-trivial throughput test are shown in the gnuplot graph in Figure 4.

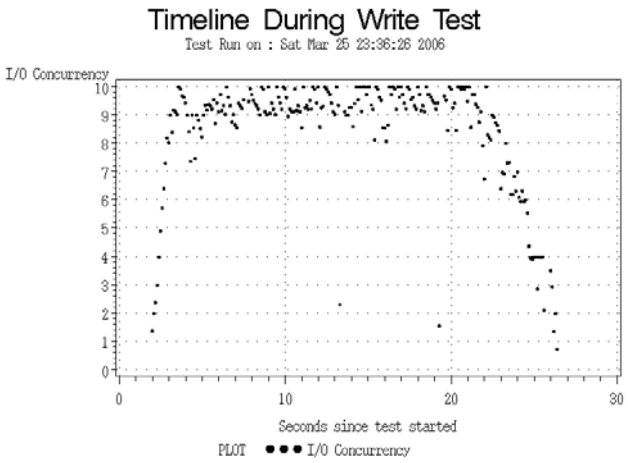

Figure 2: SAS graph of I/O concurrency during *write* test

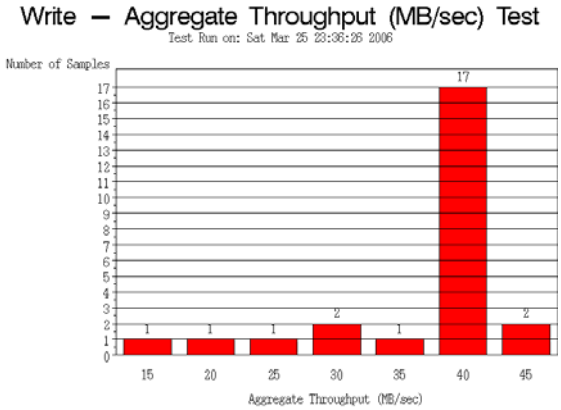

Figure 3: SAS chart showing *write* throughput histogram

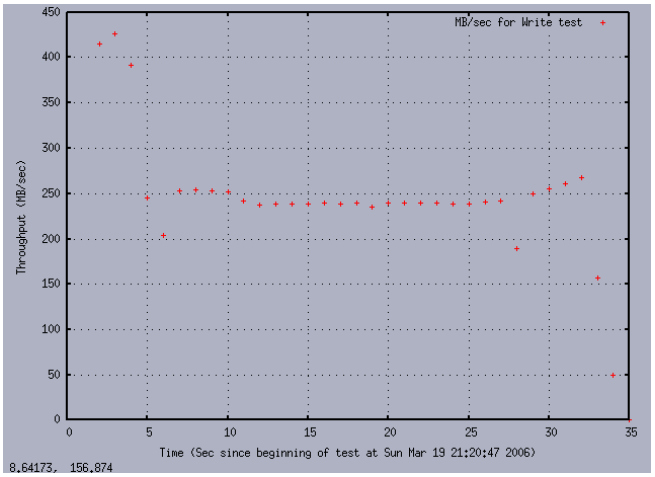

Figure 4: gnuplot graph of throughput during write test

Summary

Bertha is an extension of Bonnie, a benchmarking tool that has been in use for over a decade. It offers new features to challenge contemporary storage subsystems and provide greater insight to characterize I/O performance, as well as provide more capabilities as a benchmarking tool. It will be interesting to see if users will employ Bertha, and if so, what flaws are noted and enhancements recommended.