Datalogisk Institut
KØBENHAVNS UNIVERSITET

uFLIP: Understanding Flash IO Patterns

Luc Bouganim
Björn Þór Jónsson
Philippe Bonnet

Technical Report no. 08-13
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

uFLIP: Understanding Flash IO Patterns

Luc Bouganim
INRIA Rocquencourt
Le Chesnay, FRANCE
Luc.Bouganim@inria.fr

Björn Þór Jónsson
Reykjavík University
Reykjavík, ICELAND
bjorn@ru.is

Philippe Bonnet
University of Copenhagen
Copenhagen, DENMARK
bonnet@diku.dk

The advent of flash devices constitutes a radical change for secondary storage. For instance, writes are mapped onto program and erase operations at different granularities, and as a result the performance of writes is not uniform in time. How should database systems adapt to this new form of secondary storage? Before we can answer this question, we need to fully understand the performance characteristics of flash devices. More specifically, we want to establish what kind of IOs should be favored (or avoided) when designing algorithms and architectures for flash-based systems. In this paper, our goal is to quantify the performance of IO patterns, defined as the distribution of IOs in space and time. We define uFLIP, a component benchmark for measuring the response time distribution of flash IO patterns. We also present a benchmarking methodology which takes into account the particular characteristics of flash devices. Finally, we present the results obtained by measuring 12 flash devices, and derive a set of design hints that should drive the development of flash-based systems.

1. INTRODUCTION

Tape is dead, disk is tape, flash is disk [5]. The flash devices that now emerge as a replacement for mechanical disks are complex devices composed of flash chip(s), controller hardware, and proprietary software that together provide a block device interface via a standard interconnect (USB, IDE, or SATA). Does the advent of such flash devices constitute a radical departure from hard drives? Should the design of database systems be revisited to accommodate flash devices? Must new systems be designed differently to take full advantage of the flash devices characteristics?

In trying to answer these questions, an easy short-cut is to assume that flash devices behave as the flash chips they contain. Flash chips are indeed very precisely specified, they have interesting properties (e.g., read/program/erase operations, no updates in place, random reads equivalent to sequential reads), and many papers use their characteristics to design new algorithms [8][12][15]. The problem is that commercially available *flash devices do not behave as flash chips.* They provide a block interface, where data is read and written in fixed sized blocks. They integrate layers of software that manage block mapping, wear-leveling and error correction. As a consequence, there is a priori no reason to avoid updates-in-place on flash devices. In terms of performance, flash devices are much more complex than flash chips. For instance, block writes directed to the flash devices are mapped to program and erase operations at different granularities and as a result the performance of writes is not uniform in time. It would therefore be a mistake to model flash devices as flash chips.

So, how can we model flash devices? The answer is not straightforward because flash devices are both complex and undocumented. They are black boxes from a system's point of view.

A first step towards the modeling of flash devices is to have a clear and comprehensive understanding of their performance. The key issue is to determine the kind of IOs that should be favored (or avoided) when designing algorithms and architectures for flash-based systems.

To study this issue, we need a component benchmark that quantifies the performance of flash devices. By applying such a benchmark to a large set of current and future devices, we can start making progress towards a comprehensive understanding. While individual devices are likely to differ to some extent, the benchmark should reveal common behaviors that will form a solid foundation for algorithm and system design. In this paper, we propose such a benchmark.

So far, only a handful of papers have attempted to understand the overall performance of flash devices. Moon et al. focused on benchmarking SSD performance for typical database access patterns, but used only a single device for their measurements [9]. Myers measured the performance of database workloads over two flash devices [11]. In comparison, our benchmark is not specific to database systems. We study a variety of IO patterns, defined as the distribution of IOs in time and space. Ajwani et al. analyzed the performance of a large number of flash devices, but using ad-hoc methodology [1]. By contrast, we identify benchmarking methodology as a major challenge. It is indeed amazingly easy to get meaningless results when measuring a flash device because of the non-uniform nature of writes. Huang et al. attempted an analysis of flash device behavior, but neither proposed a complete methodology nor made any measurements [7]. Many benchmarks aim to measure disk performance (see [14] for an excellent survey and critique), but those benchmarks do not account for the non-uniform performance of writes that characterizes flash devices.

This paper makes three major contributions:

1. We define the uFLIP benchmark; a component benchmark for understanding flash device performance. uFLIP is a collection of micro-benchmarks defined over IO patterns.
2. We define a benchmarking methodology that accounts for the non-uniform performance of flash devices.
3. We apply the uFLIP benchmark to a set of 12 flash devices, ranging from low-end to high-end devices. Based on our results, we discuss a set of design hints that should drive the development of flash-based systems on current devices.

We believe that the investigation of flash device behavior deserves strong and continuous effort from the research community; an effort that we instigate in this paper. Therefore, the uFLIP software and the detailed results (more than 16 million data points) are available on a web site (www.uflip.org) that we expect to be used and completed by the community.

2. FLASH DEVICES

The uFLIP benchmark is focused on flash devices—such as solid state disks (SSDs), USB keys, or SD cards—which are packaged as block devices. While the details of flash devices vary significantly, there are certain common traits in the architecture of the flash chips and the block manager, which provides the block device abstraction, that impact their performance [3]. In this section we review those common traits.

2.1 Flash Chips

A flash chip is a form of EEPROM (Electrically Erasable Programmable Read-Only Memory), where data is stored in independent arrays of memory cells. Each array is a *flash block*, and rows of memory cells are *flash pages*. Flash pages may furthermore be broken up into *flash sectors*.

Each memory cell stores 1 bit in single-level cell (SLC) chips, or 2 or more bits in multi-level cell (MLC) chips. MLC chips are both smaller and cheaper, but they are slower and have a shorter expected life span. By default each bit has the value 1. It must be *programmed* to take the value 0 and *erased* to go back to value 1. Thus, the basic operations on a flash chip are read, program and erase, rather than read and write.

Flash devices designed for secondary storage are all based on NAND flash, where the rows of cells are coupled serially, meaning that data can only be read and programmed at the granularity of flash pages (or flash sectors).Writes are performed one page (or sector) at a time, and sequentially within a flash block in order to minimize write errors resulting from the electrical side effects of writing a series of cells.

Erase operations are only performed at the granularity of a flash block (typically 64 flash pages). This is a major constraint that the block manager must take into account when mapping writes onto program and erase commands. Most flash chips can only support up to 10^5 erase operations per flash block for MLC chips, and up to 10^6 in the case of SLC chips. As a result, the block manager must implement some form of wear-leveling to distribute the erase operations across blocks and increase the life span of the device. To maintain data integrity, bad cells and worn-out cells are tracked and accounted for. Typically, flash pages contain 2KB of data and a 64 byte area for error correcting code and other bookkeeping information.

Modern flash chips can be composed of two planes, one for even blocks, the other for odd blocks. Each flash chip may contain a page cache. The block manager should leverage these forms of parallelism to improve performance.

2.2 Block Manager

In all flash devices, the core data structures of the block manager are two maps between blocks, represented by their *logical block addresses* (LBAs), and flash pages. A *direct* map from LBAs to flash pages is stored on flash and in RAM to speed up reads, and an *inverse* map is stored on flash, to re-build the direct map during recovery. There is a trade-off between the improved read performance due to the direct map and degraded write performance due to the update of the inverse map (updates of bookkeeping information for a page may cause an erase of an entire block).

The software layer responsible for managing these maps both in RAM (inside the micro-controller that runs the block manager) and on flash is called *flash translation layer* (FTL). Using the direct map, the FTL introduces a level of indirection that allows the trading expensive writes-in-place (with the erase they incur) for cheaper writes onto free flash pages.

Each update on a free flash page, however, leaves an obsolete flash page (that contains the before image). Over time such obsolete flash pages accumulate, and must subsequently be reclaimed synchronously or asynchronously. As a result, we must assume that the cost of writes is not homogeneous in time (regardless of the actual reclamation policy). Some block writes will result in flash page writes with a minimum bookkeeping overhead, while other block writes will trigger some form of page reclamation and the associated erase. Assuming a flash device contains enough RAM and autonomous power, the flash translation layer might be able to cache and destage both data and bookkeeping information.

While the principles of the flash translation layer described above are well known, the details of its implementation and the associated trade-offs for a given flash device are not documented. Flash devices are thus black-boxes. The goal of the uFLIP benchmark is to characterize their performance.

3. THE uFLIP BENCHMARK

In this section we propose uFLIP, a new benchmark for observing and understanding the performance of flash devices. The benchmark is a set of 11 micro-benchmarks, defined in Section 3.1, which together capture the characteristics of flash devices. Benchmarking flash devices is difficult, as their performance is not uniform in time. We therefore present a benchmarking methodology in Section 3.2.

3.1 The uFLIP Micro-Benchmarks

The basic construct of uFLIP is an IO pattern, which is simply a sequence of IOs with particular characteristics. In each pattern, we refer to the i^{th} submitted IO as IO_i, and define IO_i by (a) the time at which it is submitted $t(IO_i)$, (b) its logical block address $LBA(IO_i)$, (c) its size *IOSize* and (d) a mode (read or write). We only consider direct, synchronous IO in order to bypass the host file system and to avoid interferences from the device drivers.[1]

Each micro-benchmark specifies a set of *reference patterns*, typically through formulas that define $t(IO_{i,})$ and $LBA(IO_{i,})$ with a single varying parameter. An execution of a reference pattern against a device is called a run; a collection of runs of the same reference pattern is called an experiment. We measure and record the response time for individual IOs.[2]

Note that for each pattern, we must also specify its location on the flash device (*TargetOffset*), the size of its target space (*TargetSize*), and its length (*IOCount*). Setting these parameters is part of the methodology discussed in Section 3.2.

In theory, IO patterns can be arbitrarily complex. In uFLIP, however, we focus on relatively simple reference patterns that together capture the performance of flash devices. Indeed, we observed that more complex patterns just cloud the picture.

We now define the eleven uFLIP micro-benchmarks by describing informally the sets of reference patterns and the parameter that is varied. Note that due to space constraints, we

---

[1] The lowest layer of the file system is the disk scheduler, which actually submits IO operations. The disk scheduler is, as its name indicates, designed to optimize submission of IOs to disk. Whether disk schedulers should be redesigned for flash devices is an open question; the FLIP benchmark should help in determining the answer.

[2] One could consider other metrics such as space occupation or aging. Given the block abstraction the only way to measure space occupation is indirectly through write performance measurements. Measuring aging is difficult since reaching the erase limit (with wear leveling) may take years. Measuring power consumption, however, should be considered in future work.

cannot fully specify every micro-benchmark in this paper. The complete specification of all the IO patterns and parameter settings can be found at www.uflip.org/benchmark/.

1. **Granularity**: The flash translation layer manages a direct map between blocks and flash pages, but the granularity at which this mapping takes place is not documented. The *IOSize* parameter allows determining whether a flash device favours a given granularity of IOs. The reference patterns used for this micro-benchmark are sequential reads, sequential writes, random reads, and random writes that are aligned to *IOSize* blocks; we refer to these patterns as *baseline patterns* in the remainder of this section. For the baseline patterns, IOs are contiguous in time, i.e., $t(IO_{i+1}) = t(IO_i) + rt(IO_i)$,.where $rt(IO_i)$ is the response time for IO_i.

Random: $LBA(IO_i) = TargetOffset + Random(TargetSize/IOSize) \times IOSize$

Sequential: $LBA(IO_i) = TargetOffset + i \times IOSize$

2. **Alignment:** Using a fixed *IOSize* (e.g., chosen based on the first micro-benchmark), we study the impact of alignment on the baseline patterns by introducing the *IOShift* parameter and varying it from 0 to *IOSize*.

Random: $LBA(IO_i) = TargetOffset + Random(TargetSize/IOSize) \times IOSize + IOShift$

Sequential: $LBA(IO_i) = TargetOffset + i \times IOSize + IOShift$

3. **Locality:** We study the impact of locality on the random baseline patterns, by varying *TargetSize* down to *IOSize.*

4. **Circularity:** We study the impact of circularity on the sequential baseline patterns by varying *TargetSize* from *IOSize* to $IOCount \times IOSize / 2$.

$LBA(IO_i) = TargetOffset + (i \times IOSize)\ mod\ TargetSize$

5. **Partitioning:** The partitioned patterns are a variation of the sequential baseline patterns. We divide the target space into *P* partitions which are considered in a round robin fashion; within each partition IOs are performed sequentially. This pattern represents, for instance, a merge operation of several buckets during external sort. If we denote the partition size by $PS = TargetSize/P$, the partition written to at step *i* as $P_i = i\ mod\ P$, and the offset within the partition as $O_i = \lfloor i/P \rfloor\ mod\ PS$, then:

$LBA(IO_i) = TargetOffset + (PS \times P_i + O_i) \times IOSize$

6. **Order:** The order patterns are another variation on the sequential patterns, where logical blocks are addressed in a given order. For the sake of simplicity, we consider a linear increase (or decrease) in the LBAs addressed in the pattern, determined by a linear coefficient *k*. We can thus define a) patterns with increasing LBAs ($k > 1$) or decreasing LBAs ($k < 0$), or b) in-place patterns ($k = 0$) where the LBA remains the same throughout the pattern. These mapping are simple, yet important and representative of different algorithmic choices: for example, a reverse pattern ($k = -1$) represents a data structure accessed in reverse order when reading or writing, the in place pattern is a pathological pattern for flash chips, while an increasing LBA pattern represents the manipulation of a pre-allocated array, filled by columns or lines.

$LBA(IO_i) = TargetOffset + i \times k \times IOSize$

7. **Parallelism:** Since flash devices include many flash chips (even most USB keys contain two flash chips), we want to study how they support overlapping IOs. We divide the target space into *D* (possibly overlapping) subsets, each one accessed by a process executing the same baseline pattern. We vary the parameter *D* to study how a flash device supports parallelism and thus how asynchronous IO should be scheduled, and how parallelism should be managed. We omit details due to lack of space.

8. **Mix Read/Write:** We compose two baseline patterns, one with reads and the other with writes. Both patterns are either sequential or random. We vary the ratio of reads to writes to study how such mixes vary from the baselines. We omit details due to lack of space.

9. **Mix Sequential/Random:** We compose two baseline patterns, one with sequential IOs and the other with random IOs. Both patterns consist either of reads or writes. We vary the ratio of sequential to random IOs to study how such mixes vary from the baselines. We omit details due to lack of space.

10. **Pause:** This is a variation of the baseline patterns, where IOs are not contiguous in time. We introduce a parameter *Pause*, expressed in msec., and vary the *Pause* parameter to observe whether potential asynchronous operations from the flash device block manager impact performance.

$t(IO_{i+1}) = t(IO_i) + rt(IO_i) + Pause$

11. **Bursts:** This is a variation of the previous micro-benchmark, where a single pause of a fixed length is introduced after a fixed number of IOs, rather than every IO. The *Pause* parameter is then varied to study how potential asynchronous overhead accumulates in time. We omit details due to lack of space.

Even though uFLIP is not a domain-specific benchmark, it should still fulfill the four key criteria defined in the Benchmarking Handbook: portability, scalability, relevance and simplicity [6]. Because uFLIP defines how IOs should be submitted, uFLIP has no adherence to any machine architecture, operating system or programming language: uFLIP is portable. Also, uFLIP does not depend on the form of flash device being studied, we have indeed run uFLIP on USB keys, SD cards, IDE flashes and SSD drives: uFLIP is scalable. We believe uFLIP is relevant to algorithm, system and flash designers because the 11 micro-benchmarks are based on flash characteristics as well as on the characteristics of the software that generates IOs. It is neither designed to support decision making nor to reverse engineer flash devices. Whether uFLIP satisfies the criteria of simplicity is a bit trickier. The benchmark definition itself is quite simple. Indeed, we reduced an infinite space of IO patterns down to 11 micro-benchmarks that define how IOs are submitted using very simple formulas. Benchmarking flash devices, however, is far from simple because their performance is not uniform in time. The methodology we present in the next section addresses this issue. Furthermore, our decision to measure response time for each submitted IO means that the benchmark results are very large and analysing those results is not straightforward. In the Demonstration Section, we present a visualization tool that facilitates result analysis.

3.2 Benchmarking Methodology

The fact that response time is non-uniform in time is a real challenge when measuring flash performance. First, the initial state of the device matters. Second, each micro-benchmark should be large enough to capture the variations representative of the device under study. Third, consecutive micro-benchmark runs should not interfere with each other.

Initial State: Ignoring the initial state of a flash device leads to meaningless performance measurements. Out-of-the-box, the Samsung SSD had excellent random write performance (around 1 msec for a 16KB random write, compared to around 8 msec for other SSDs). After we randomly wrote the entire 32GB of flash,

however, the performance decreased by almost an order of magnitude.

In order to obtain repeatable results, we need to run the micro-benchmarks from a well-defined initial state, independent from the complete IO history. Since flash devices only expose a block device API, we cannot erase all blocks and get back to factory settings. Because flash devices are black boxes, we cannot know their exact state. We thus make the following assumption: *Writing the whole flash device completely defines its state*. The rationale is that the direct and indirect maps managed by the FTL are filled and well-defined.

We propose to enforce an initial state for the benchmark, by performing random IOs of random size (ranging from 0.5KB to the flash block size) on the whole device. This method is quite slow, but stable since only sequential writes disturb the state significantly. We alleviate this problem by grouping sequential writes to distinct target spaces (specified by *TargetOffset*) when running the micro-benchmarks. The alternative—performing a complete rewrite of the device using sequential IOs of a given size—is faster but less stable, as random writes, badly aligned IOs, or IOs of different sizes, modify the initial state.

Start-up and Running Phases: Consider a device where the first 128 random writes are very cheap (400 µsec), and where the subsequent random writes oscillate between very cheap (400 µsec) and very expensive (27 msec). If we run the Mix Read/Write micro-benchmark with an *IOCount* of 1024, we will get meaningless results when the ratio of random writes is lower than 1/8 (because then our measurements only capture the initial, very cheap random writes). If we are not careful, we might even conclude that mixing reads with a few writes has an excellent impact on performance.

We propose a two-phase model to capture response time variations within the course of a micro-benchmark run. In the first phase, that we call *start-up phase*, response time is cheap (because expensive operations are delayed). In the second phase, that we call *running phase*, response time is oscillating between two or more values. We characterize each device by two parameters: *start-up*, which defines the number of IOs for the start-up phase, and *period*, which defines the number of IOs in one oscillation in the running phase. In order to measure *start-up* and *period*, we run all four baseline patterns (SR, RR, SW and RW) with a very large *IOCount.* We can then identify the two phases for each pattern (the start-up phase may not be present) and derive upper bounds across the patterns for *start-up* and *period.*

The impact of this two-phase model on the benchmarking methodology is twofold. First, for each experiment we must adjust *IOCount* to capture both the start-up phase and the running phase. Second, we must ignore the start-up phase when summarizing the results of each run, so that we can use a statistical representation (min, max, mean, standard variation) to represent the response times obtained during the running phase.

No interference: Consecutive benchmark runs should not interfere with each other. Consider a device that implements an asynchronous page reclamation policy. Its effects should be captured in the running phase defined above. We must make sure, however, that the effect of the page reclamation triggered by a given run has no impact on subsequent, unrelated runs.

To evaluate the length of the pause between runs, we rely on the following experiment. We submit sequential reads, followed by a batch of random writes, and sequential reads again. We count the number of sequential reads in the second batch which are affected by the random writes. We use this value as an upper bound on the pause between consecutive runs.

4. FLASH DEVICE EVALUATION

In this section, we report on our experimental evaluation of a range of flash devices, using the uFLIP benchmark. It was quite difficult to select a representative and diverse set of flash devices, as a) the flash device market is very active, b) products are not well documented (typically, random write performance is not provided!), and c) in fact, several products differ only by their packaging. We eventually selected 13 different devices[3], from low-end USB keys or SD cards to high-end SSDs. While we ran the entire uFLIP benchmark for all the devices, we only present results for six representative devices listed in Table 1. Detailed information and measurements for all the devices can be found at http://www.uflip.org/results.html.

We ran the uFLIP benchmark on an Intel Celeron 2.5GHz processor with 2GB of RAM running Windows XP. We ran each micro-benchmark using our own software package, FlashIO available at http://www.uflip.org/flashio.html). We ran uFLIP three times on each device; the differences in performance were typically within 5%.

4.1 Benchmark Results

Due to space limitations, we only present highlights of our results.

As mentioned in Section 3.2, we first filled each device with random writes of random size, and then ran the baseline patterns with large *IOCount* to measure *startup* and *period* for each device. Figures 1 and 2 show the most interesting traces obtained on the MTRON SSD and on the Kingston USB key, for RW and SW respectively. The *x*-axis shows the time in units of IO operations, while the *y*-axis shows the cost of each operation in msec (note the logarithmic scale). In Figure 1, we can easily distinguish between the startup phase and the running phase, while in Figure 2 there is no startup phase. In Figure 1, the dashed line represents the running average of response time, including the startup phase measurements, while the solid line represents the running average of response time, excluding the start-up phase measurements. As expected, excluding the startup phase measurements results in a faster and more accurate representation of response time.

With respect to startup and running phases, we can basically divide the set of tested devices in two classes. The Memo GT, and MTRON SSDs both have a startup phase for random writes followed by oscillations with very small period. They do not show startup or oscillations for SR, RR and SW. For these devices, care should be taken when running experiments that involves a small number of RW, typically Mix Read/Write since the startup phase should be scaled-up according to the number of RW. The other 10 devices have no startup phase but show small oscillations for RR, larger one for SW and sometimes remarkable oscillations for RW (with some impressive variations between 0.25 and 300 msec!). For simplicity, we fixed *IOCount* to 1024 for all the tested devices and checked manually the stability in the running phase.[4]

Let us now study the performance of the Granularity micro-benchmark where *IOSize* is varied. We generally expect reads to be cheaper than writes because some writes will generate erase operations, and we also expect random writes to be more

[3] At the time of submission, we were still waiting for the recently released Flash PCI card from Fusion-IO, advertised as reaching throughput of 600MB/s for random writes. We plan to publish the benchmarking results on our web-site before the end of October.

[4] The automatic determination of the smallest correct *IOCount* is left for future work.

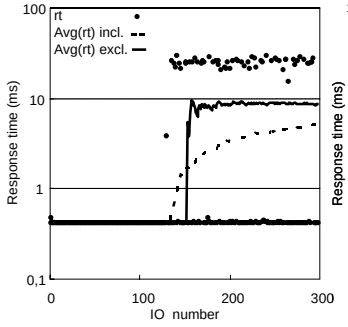

Fig. 1: Starting and running phase for Memo GT

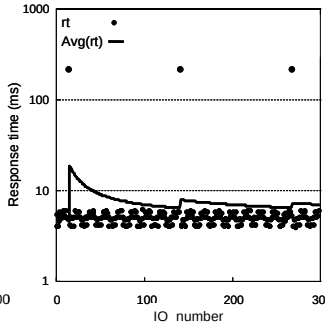

Fig. 2: Running phase for Kington DT

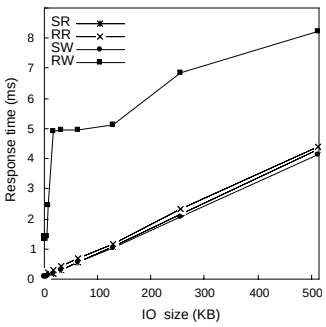

Fig. 3: Granularity for Mtron

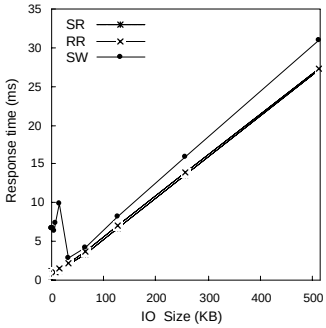

Fig. 4: Granularity for Kingston DT

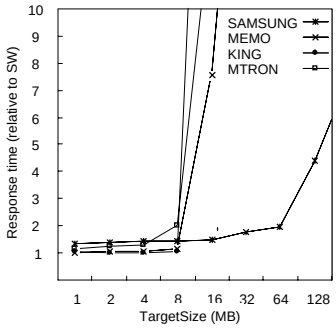

Fig. 5: Locality for four devices

	Startup	Period	SR	RR	SW	RW	Pause	Locality	Partitioning	Reverse	InPlace
	(IOs)		(ms)					(MB)	(P)	(k=-1)	(k=0)
MemoGT 32GB SSD ($943)	50	20	0,3	0,4	0,3	5	5	8 (=)	8 (=)	=	=
Mtron 32GB SSD ($407)	130	20	0,4	0,5	0,4	8	8	8 (x2)	2 (x2)	=	=
Samsung 32GB SSD ($517)	0	100	0,5	0,7	0,4	18		64 (x2)	32 (x2)	x2	x0.6
Transcend 4GB IDE Module ($61)	0	30	0,8	0,8	1,1	18		4 (x3)	16 (x2)	x2	x11
Kingston DTHX 8GB USB ($153)	0	150	1,7	1,8	2,0	270		8 (=)	4 (=)	x3	x3
Kingston DT 4GB USB ($17)	0	280	1,9	2,2	2,9	219		8 (=)	4 (=)	=	=

Table 1: Results summary

expensive than sequential writes as they should generate more erases. Figure 3 shows the response time (in msec) of each IO operation for the MemoGT SSD. Three observations can be made about this figure. First, all reads and sequential writes are very efficient; their response time is linear with a small latency (about 70 µsec for SR/SW and 115 µsec for RR). Second, for rather large random writes, the response time is much higher, at least 5 msec; note that, similar to Figure 1, the cost of random writes alternates between cheap writes (of similar cost to sequential writes) and extremely expensive erase operations. Third, small random writes are serviced much faster; apparently due to caching as four writes of 4KB take about as much time as two writes of 8KB and one write of 16KB. In comparison, Figure 4 shows the response time for the Kingston DT USB key. In this figure, the response time of random writes is omitted, as it is more than 200 msec. As the figure shows, for this device the cost of sequential writes is affected strongly by the IO granularity, as smaller writes incur a significantly higher cost than writes of 32KB. Comparing the two devices, we observe that while random writes are up to a factor of five times slower than the other operations on the MemoGT, they are one or two orders of magnitude slower for the Kingston DT USB key. This is undoubtedly due to more advanced hardware and FTL on the MemoGT SSD. The remainder of these experiments was run with IO sizes of 32KB.

To give a short outline of the results of the micro-benchmarks that we do not cover in detail (typically because their results were predictable), we observed the following. Using IO granularities that were not aligned with flash page sizes resulted in most cases in performance degradation, as did unaligned IO requests. Composite patterns of random reads and writes, or sequential and random writes, did not affect the overall cost of the workloads. Circularity does not affect performance, until the area written to is so small that the writes become in-place writes (see below for the effect of in-place writes). Finally, we did not notice any significant improvement while submitting IOs in parallel.

Moving on to the more interesting results, we first consider the effect of locality on random writes. Figure 5 shows the response time of random writes (relative to sequential writes) as the target size grows from very small (local) to very large (note the logarithmic *x*-axis). Our expectation was that doing random writes within a small area might improve their performance. The figure verifies this intuition, as random writes within a small area have nearly the same response time as sequential writes. The figure shows, however, that the exact effect of locality varies between devices, both in terms of the area that the random writes can cover, and in terms of their relative performance.

Table 1 succinctly summarizes the remainder of the experiments; we will discuss the result columns from left to right. First, SR, RR, SW, RW indicate the cost of a corresponding IO operation of 32KB. These columns show that there is a large difference in performance between the USB keys and the other devices, but also between low-end and high-end SSDs. For the high-end SSDs, even the random write performance is quite good. In fact, as we explore more results, the high-end SSDs distinguish themselves further from the rest.

The fifth column of Table 1 indicates the effect of inserting pauses into a random write workload. No value indicates that this had no effect, which in turn indicates that no asynchronous page reclamation is taking place. For the high-end SSDs, however, inserting a pause improves the performance of the random writes to the point where they behave like sequential writes. Interestingly, the length of the pause when that happens is roughly the time required on average for a random write. Thus, no true response time savings are seen by inserting this pause, as the total workload takes the same overall time regardless of the length of the pause. A similar effect is seen with the Burst micro-benchmark.

The sixth column of Table 1 summarizes the effect on locality, which we already explored with Figure 5; it shows the size of "locality area" in MB and, in parentheses, the cost of random writes within that area relative to sequential writes.

The seventh column of Table 1 summarizes a similar effect for the Partitioning micro-benchmark. The goal of that experiment was to study whether concurrent sequential write patterns to many partitions degrade the performance of the sequential writes. The column shows the number of concurrent partitions that can be written to without significant degradation of the performance, as well as the cost of the writes relative to sequential writes to a single partition. Note that when writing to more partitions than indicated in this column, the write performance degrades significantly.

Finally, the last two columns show the cost of the reverse and in-place patterns, compared to the cost of sequential writes. As the columns show, the effect of the in-place pattern, in particular, varies significantly between devices, ranging from time savings of about 40% for the Samsung SSD, to a performance degradation of an order of magnitude for the Transcend IDE module.

4.2 Discussion

The goal of the uFLIP benchmark is to facilitate understanding of the behavior of flash devices, in order to improve algorithm and system design against such devices. In this section we have used the uFLIP benchmark to explore the characteristics of a large set of representative devices (although we have presented a limited set of results due to space constraints). From our results, we draw three major conclusions.

First, we have found that with the current crop of flash devices, their performance characteristics can be captured quite succinctly with a small number of performance indicators (the ones shown in Table 1 and a few more).

Second, we observe that the performance difference between the high-end SSDs and the remainder of the devices, including low-end SSDs, is very significant. Not only is their performance better with the basic IO patterns, but they also cope better with unusual patterns, such as the reverse and in-place patterns. Unfortunately, the price label is not always indicative of relative performance, and therefore designers of high-performance systems should carefully choose their flash devices.

Finally, based on our results, we are able give the following design hints for algorithm and system designers:

Hint 1: Flash devices do incur latency. Therefore, larger IOs are generally beneficial, even for read operations.

Hint 2: Block size *should (currently) be 32KB.* Based on the first hint, large block sizes are beneficial for writes, while an application of the famed five minute rule [4] says 4KB pages are beneficial for reads, based on prices and capacities of the high-end devices we studied. We therefore believe that 32BK is a good trade-off for those high-end devices.

Hint 3: Blocks should be aligned to flash pages. This is not unexpected, based on flash characteristics, but we have observed that the penalty paid for lack of alignment is quite severe.

Hint 4: Random writes should be limited to a focused area. Our experiments show that random writes to an area of 4–16MB perform nearly as well as sequential writes.

Hint 5: Sequential writes should be limited to a few partitions. Concurrent sequential writes to 4–8 different partitions are acceptable; beyond that performance degrades to random writes.

Hint 6: Neither concurrent nor delayed IOs improve the performance. Due to the absence of mechanical components, IO scheduling is not improved through abundance of pending asynchronous IOs. Furthermore, introducing a pause does not affect total response time.

5. CONCLUSION

The design of algorithms and systems using flash devices as secondary storage should be grounded in a comprehensive understanding of their performance characteristics. We believe that the investigation of flash device behavior deserves strong and continuous effort from the community: uFlip and its associated benchmarking methodology should help define a stable foundation for measuring flash device performance. By making available online (at www.uflip.org) the benchmark specification, the software we developed to run the benchmark, and the results we obtained on 12 devices, our plan is to gather comments and feedback from researchers and practitioners interested in the potential of flash devices. Future work includes automatic tuning each run's length, to ensure that the start-up period is omitted and the running phase captured sufficiently well to guarantee given bounds for the confidence interval, while minimizing .the number of IOs issued.

6. REFERENCES

[1] Ajwani, D., Malinger, I., Meyer, U., Toledo, S. Characterizing the performance of flash memory storage devices and its impact on algorithm design. *Proc. Workshop on Experimental Algorithms (WEA)*, Provincetown, MA, USA, 2008.

[2] Anderson, E., Kallahalla, M., Uysal, M., Swaminathan, R. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. *Proc. USENIX Conf. on File and Storage Technologies*, San Francisco, CA, USA, 2004.

[3] Gal, E., Toledo, S. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2), 2005.

[4] Graefe, G. The five-minute rules twenty years later, and how flash memory changes the rules. *Proc. Data Management on New Hardware (DaMoN)*, Beijing, China, 2007.

[5] Gray, J. Tape is dead, disk is tape, flash is disk, RAM locality is king. *Pres. at the CIDR Gong Show*, Asilomar, CA, USA, 2007.

[6] Gray, J. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[7] Huang, P.-C., Chang, Y.-H., Kuo, T.-W., Hsieh, J.-W., Lin, M. The behavior analysis of flash-memory storage systems. *Proc. IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, Orlando, FL, USA, 2008.

[8] Lee, S.-W., Moon, B. Design of flash-based DBMS: An in-page logging approach. *Proc. ACM SIGMOD*, Beijing, China, 2007.

[9] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., Kim, S.-W. A case for flash memory SSD in enterprise database applications. *Proc. ACM SIGMOD*, Vancouver, BC, Canada, 2008.

[10] Leventhal, A. Flash storage memory. *Communications of the ACM*, 51(7), 2008.

[11] Myers, D. On the use of NAND flash memory in high-performance relational databases. Master's thesis, MIT, 2008.

[12] Nath, S., Kansal, A. FlashDB: Dynamic self-tuning database for NAND flash. *Proc. Information Processing in Sensor Networks*, Cambridge, MA, USA, 2007.

[13] Open Source Development Lab. IOmeter. www.iometer.org.

[14] Trayger, A., Zadok, E., Joukov, N., Wright, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2), 2008.

[15] Wu, C.-H., Kuo, T.-W. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 6(3), 2007.

7. DEMONSTRATION

In the interest of the research community, we will make our open source software package available, as well as our complete database of measurements. The web-site is hosted at www.uflip.org. In this section, we propose a demonstration of our software package and the measurement database.

7.1 Software Architecture

The uFLIP benchmark software consists of four components.

- The uFLIP Interface, which allows the user to choose settings for individual micro-benchmarks, and to configure the initial state of the flash device.
- The FlashIO workload generator, which executes the micro-benchmarks defined by the uFLIP Interface, and stores the results into a database.
- The uFLIP Database, which is divided into two parts: a core area for verified benchmark results, and a holding area for transient or non-verified results.
- The uFLIP Visualizer, which presents results from the uFLIP Database.

The first three components are rather straightforward. We now describe the uFLIP Visualizer in more detail.

The interface of the uFLIP Visualizer prototype is shown in Figure 6. Once a device has been selected, a "dashboard" is shown, which is a matrix of micro-benchmarks and IO modes (random reads, sequential reads, random writes, sequential writes) is displayed. For each such combination, interesting results are indicated with a red color.

From this dashboard, individual results can then be explored by viewing graphs similar to Figure 3. Furthermore, when more detail is desired, graphs such as that in Figure 1 can be viewed. Through this navigation, results for individual devices can be interactively explored.

7.2 Demonstration Presentation

During the 10 minute presentation, we propose to demonstrate the use of the uFLIP Interface and the uFLIP Visualizer (running the uFLIP benchmark is too time consuming). First, we will show how to manipulate the settings of the interface, in order to run the benchmark and facilitate exploration of the flash device's properties. Second, we will demonstrate the navigation of the results database, using one of the high-end devices reported on in this paper.

7.3 Off-line Demonstration

Off-line demonstration to individual conference attendees is always most entertaining and informative. In order to facilitate discussions and explorations, the uFLIP Interface will include a "demo" button, which can be used to select settings that allow running the benchmark in a reasonable time-frame, even for USB keys of conference attendees. Once the results have been stored to the holding area, they can be explored interactively using the uFLIP Visualizer.

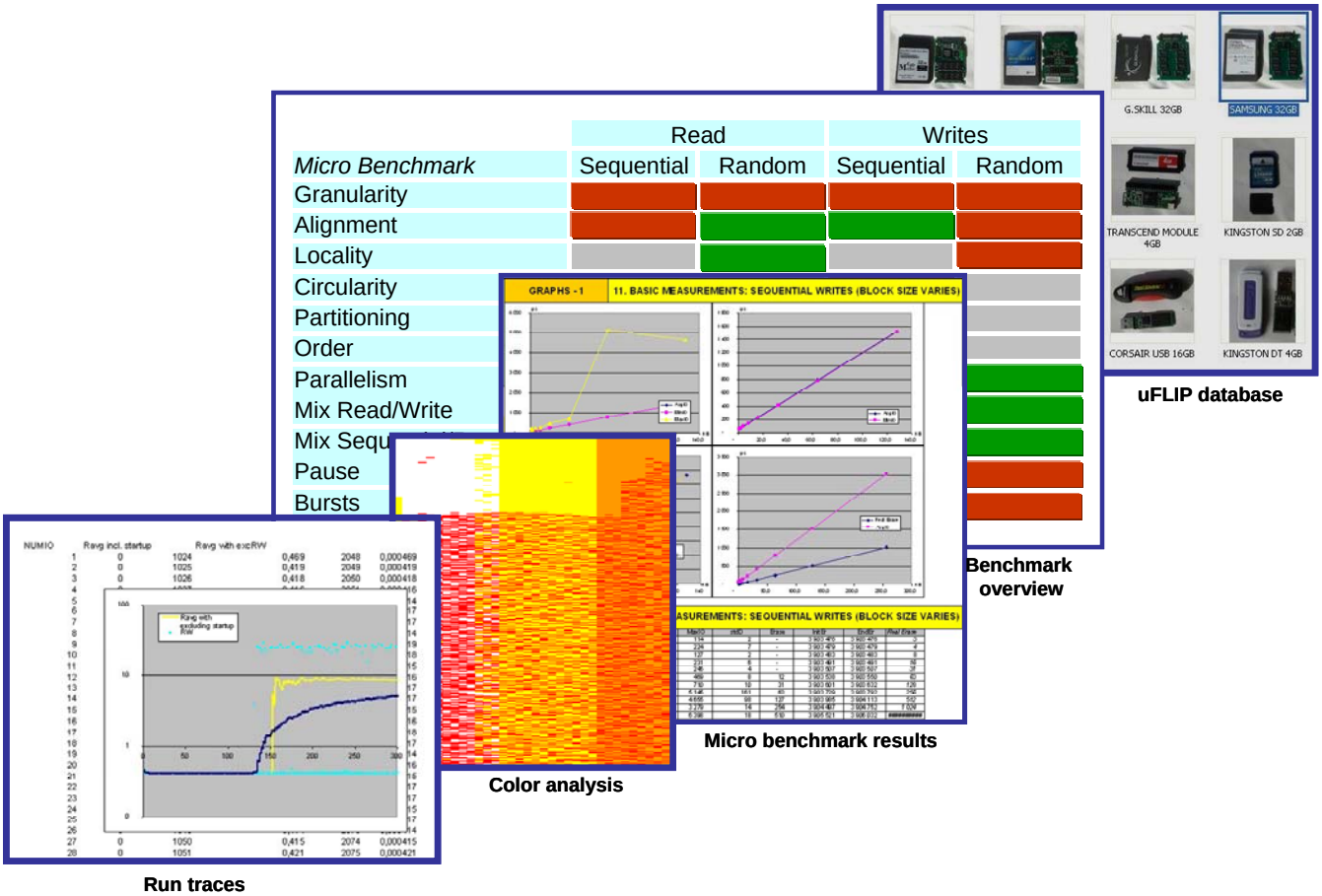

Fig 6. Demonstration prototype screenshots