Extrapolation Pitfalls When Evaluating Limited Endurance Memory

Rishiraj A. Bheda, Jesse G. Beu, Brian P. Railing, Thomas M. Conte,
Georgia Institute of Technology (*rbheda3, jbeu3, brian.railing, conte @gatech.edu*)

Abstract

Many new non-volatile memory technologies have been considered as a future scalable alternative to DRAM. Memory technologies such as MRAM, FeRAM, PCM have emerged as the most viable alternatives. But these memories have limited wear endurance. Practically realizable main memory systems employing these memory technologies are possible only if the wear across these memories is reduced as well as uniformly distributed. Limited endurance has resulted in extensive wear leveling research with the goal of uniformly distributing write traffic throughout available physical memory. Basic support for wear leveling is already present in existing systems, in the form of operating system paging. The Operating System (OS) changes virtual to physical translations over time. As a result, write traffic is naturally spread out. Proper evaluation of the need for wear leveling as well as the impact of the corresponding technique must take this phenomenon into account. Ignoring the effect of OS paging mechanism can result in highly inaccurate memory lifetime extrapolations. We demonstrate through simulation results, the effects of inaccurate extrapolations in the absence of OS modeling. Accurate memory lifetime simulation can take from many months to years. Although sampling techniques are commonly employed for speedup, our results show that naïve extrapolation techniques can lead to wildly different lifetime estimates. We show how sampling can be accurately applied by accounting for the different components in the write stream observed by main memory. Finally, we present a heuristic to quickly estimate memory lifetime for a given application.

Index Terms—PCM, Wear Leveling, Operating System

I. INTRODUCTION

Existing DRAM designs will not be able to scale due to power and feature size limitations beyond a few generations. As a result, non-volatile memory technologies, such as Magneto-Resistive Memory [13], Ferro-Electric Memory [13], and Phase Change Memory (PCM) [1] are being evaluated as alternatives for the future. However, many non-volatile memory technologies suffer from limited write endurance, like Flash Memory [1]. In the wake of this problem, many research studies have emerged to address the issue of limited write endurance for such memories. These can be broadly categorized into three classes of solutions: write filtering [14], wear leveling [2] and write prevention [3]. It has been widely accepted that limited endurance memory systems will likely have a combination of several such techniques to increase their lifetime.

Evaluation of such systems is challenging for several reasons. Simulation time for a single execution run is already prohibitively slow for large applications and/or large-scale systems. This problem is greatly exacerbated when measuring Mean Time to Failure (MTTF), which is often in years. Further, designers of such systems are trying to evaluate future systems with projected future memory densities, resulting in simulated systems that have memories several times larger than the native system on which, the simulation is running on. Out of necessity, designers are forced to take several short cuts when estimating MTTF of limited endurance memory systems. These include the use of reduced benchmark traces to reduce the simulated execution time for a single benchmark run, extrapolating years worth of execution from a single process's results, and simulating with a reduced memory size.

In this work, we show the impact that these commonly employed extrapolation techniques have on MTTF estimates. Specifically we show that the operating system (OS) plays a prominent role both in single-process and memory-size extrapolation, but is often ignored. Meanwhile, estimates not taking the OS into account can be off by several orders of magnitude due to paging effects.

The contributions of this work are as follows:

- Highlight the impact of the operating system and importance of inter-process simulation with respect to write endurance evaluation. We show that the operating system's paging mechanism acts as a natural wear-leveling mechanism over time and must be taken into consideration.
- Propose a simple heuristic for estimating write endurance from a single run of a benchmark *for normal benchmark execution* (i.e., in the absence of an attack) as an accurate alternative to naïve single-process to multiple-process extrapolation.
- Demonstrate the importance of memory size selection when evaluating wear endurance. We show that care must be taken when extrapolating MTTF from smaller memory sizes for estimating MTTF at larger memory sizes due to OS intra-process effects, such as page-replacements, and wear distribution across larger memories.
- We also present accuracy result for reduced trace execution and propose a simple sampling methodology as an alternative to commonly employed 'skip X and execute Y instruction' techniques. Our results show that non-sampling bias does not have a significant impact on MTTF and thus no warm-up techniques between samples need be applied.

The remainder of the paper is organized as follows. We first give an overview of the leading limited endurance memory

technologies and discuss prior work for improving MTTF, with emphasis on wear-leveling mechanisms. This is followed by a section dedicated to the impact OS paging has on limited endurance memory research. Results are then presented showing the impact of naive extrapolation for single to multi-process extrapolation and small to large memory extrapolation. Correct extrapolation methodologies are presented that take the OS into consideration. For completeness, the impact of reduced trace to full-run extrapolation is demonstrated, as well as an accurate sampling methodology alternative. We provide a heuristic to quickly estimate maximum achievable memory lifetime for a given application and show results for the same.

II. Limited Endurance Memory and Related Research

The ever-increasing need for higher main memory capacity has driven the search for a memory technology that is scalable, denser, and faster with every generation while preferably consuming less energy. While many new non-volatile memory technologies are emerging as viable replacements for DRAM, they come with the downside of limited write capability. The following subsections will discuss some of the leading technologies, and solutions proposed in the literature for improving wear endurance for write limited memory.

A. Magneto-Resistive Memory

The MRAM (Magneto-resistive Random Access Memory) concept is based on the TMR (Tunneling Magneto-Resistance) effect. In each memory cell, there is an MTJ (Magnetic Tunnel Junction), which in its simplest form is a MIM (Metal-Insulator-Metal) structure with ferromagnetic electrodes. By applying a small bias voltage between the electrodes, a tunnel current can flow. The tunnel resistance depends on the relative orientation of the magnetization vector of the ferromagnetic electrodes. If the magnetization vectors are parallel, then the tunnel resistance is small. If the magnetization vectors are anti parallel, then the tunnel resistance is large. In an MRAM MTJ the direction of magnetization of one of the two ferromagnetic electrodes is fixed, whereas the magnetization of the other electrode can be switched via on-chip currents (free magnetic layer). By exposing the MTJ to an external magnetic field a hysteresis loop is formed. The two stable states of the MTJ correspond to a '0' or a '1' in the absence of an external magnetic field. The information stored in the memory cell is read by determining the tunnel resistance of the MTJ. In order to write information into the MRAM cell, current is passed through the two orthogonal metal lines, so that the magnetic field at the intersection is large enough to flip the magnetization of the free magnetic layer. After a limited number of writes, the free magnetic layer can no longer be magnetized. MRAM can tolerate ~10^{15} writes per cell [13].

B. Ferro-Electric Memory

In FeRAMs (Ferro-electric Random Access Memories) the remnant polarization of a ferro-electric thin film is used for information storage. With an externally applied electric field the polarization can be switched and the information is retained even if the external field is removed. In the absence of an external field, the polarization has two distinct stable states. In order to read data from a FeRAM memory cell the ferro-electric capacitor is connected to the pre-charged bit line. The plate line then is pulsed to a certain potential and thus a voltage is applied across the serially connected cell and bit line capacitance. Depending on the polarization state of the memory cell, a smaller amount of charge ("non-switching" case, '0') or a larger amount of charge ("switching" case, '1') flows onto the bit line capacitance. This charge is transformed into a read voltage and two different signal levels are obtained accordingly. Applying a positive or negative voltage across the memory capacitor programs the FeRAM memory cell. This programming voltage sets the polarization to either of the two possible states. Constant change in polarization causes material degradation resulting in a limited write capability. FeRAM can tolerate ~10^{9} -10^{10} writes per cell [13].

C. Phase Change Memory

Phase Change Memory (PCM), is a form of resistive memory that has gained a lot of interest due to its scalability and low power. PCM is a type of non-volatile memory that uses the unique behavior of chalcogenide ($Ge_2Sb_2Te_5$ or *GST*) glass for storing data bits. The state of this material can be altered between an amorphous and a crystalline state by application of a current pulse that heats and cools the material either slowly or quickly, thus changing the state and resistance of the material. The resistance of the phase change material determines the data value stored by a PCM cell. However, this frequent phase change due to heating and cooling of the material causes it to wear out and eventually get stuck in an undefined state. Thus PCM cells can tolerate ~10^{7}-10^{9} writes per cell [1].

D. Improving Lifetime of Limited Endurance Memory

With the emergence of so many limited endurance memory technologies, there has been a surge of research in recent years on ways to improve MTTF for such devices. While most research focuses primarily on PCM, most mechanisms do not take advantage of any specific PCM details, and thus are in general applicable to any limited endurance memory. We identify three categories to give a higher-level perspective of research in this area. They are:

1. Write Filtering (Caching) – System Level [14]
2. Wear Leveling (Distribution of writes across physical memory) – Memory Level [2]
3. Write Prevention (Comparing data and writing only as necessary, and prevention of malicious write attacks) – Data Dependent [3], PCM Security [6,7]

Figure 1, shows the most common arrangement of these

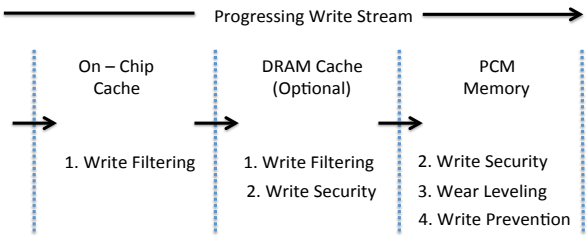

Figure 1 : System level overview of wear-leveling research

components in a memory hierarchy. The memory reference stream from the processor is filtered by the on-chip cache hierarchy. If the last-level cache is write-back, temporal locality will prevent many repeated accesses from being seen at main memory. Upon leaving the memory controller and entering the main-memory module, another level of caching may be encountered, such as a DRAM cache. This has a multitude of benefits, including latency reduction and write endurance protection and is explored in detail by Qureshi et al. in [14]. Additionally, a structure for malicious attack detection and handling may be present in this stage (monitoring traffic either before, after, or in parallel with the DRAM cache) [6,7].

In preparation for accesses that will reach the main memory, wear leveling is employed in an effort to distribute memory access density evenly across the address space. In doing so, write-heavy locations are spread out, avoiding non-uniform wear on the cells. This results in a shift in mean time to failure (MTTF) estimates, moving MTTF closer to the theoretical maximum.

A final preventive measure can be applied to reduce the number of bits being modified on a write. These techniques are data-dependent, relying on comparisons between the previously stored values and the soon-to-be-written data, based on minimizing the Hamming distance between the two data values.

III. Common Extrapolation Pitfalls

A. *Wear Leveling, Multiple Processes and the Operating System*

The primary objective of wear leveling research is to achieve an even distribution of writes across physical memory in order to avoid early failure from premature wear out of some cells. During the execution of a program, certain memory locations are more heavily stressed than others and hence tend to wear out faster. For example, lower-order bits have a tendency to change more frequently than high-order bits within a machine word [3], and lower-numbered blocks within a page frame tend to be written to more heavily than other blocks [5], and even at the page granularity some pages are much more active than others.

It is a common practice to use a single execution of a benchmark for evaluation of memory behaviors. On the surface this does not seem to pose any problems, so time extrapolation for MTTF estimates seems reasonable. However, for write endurance, specifically wear leveling, the OS plays a significant role in lifetime estimation because it induces natural wear leveling *across multiple runs*. This is because the OS maintains a mapping between virtual and physical addresses that changes over time, both as a program executes (intra-process effects) and during process termination and creation (inter-process effects). In a real system both intra- and inter-process effects migrate hot locations, at the page granularity, during a memory's lifetime. While there is still potentially wear-leveling opportunity at the intra-page level, the natural wear leveling due to OS paging already improves lifetime by several orders of magnitude for applications that do not pin virtual-to-physical page mappings, as our simulation results will show. Even when an OS is modeled, a single run will not capture inter-process effects. As a result, this kind of extrapolation causes gross under-estimation of baseline MTTF for limited endurance memories.

Additionally, there is another inter-process effect from the OS when multiple processes execute simultaneously. As each program starts, the operating system allocates a set of available physical pages to the different processes. During execution, the OS will continue to provide each process with a reasonable set of physical pages; as the working sets of an individual process change, the OS will provide new pages or reclaim infrequently used pages from one program's use for reuse by other contending processes. This process of page allocation and reclamation can change the set of physical pages of a process even during single run of a workload, further removing a single, isolated execution from being representative of actual behavior on a real system.

B. *Proper Main-Memory Sizing*

A majority of the write endurance community has ignored memory size as a concern, either modeling only a single memory block [3] or simulating memories much smaller than the projected future memory sizes[6]. Meanwhile, PCM memory is projected to provide upwards of 32GB of memory or more as the technology matures [1, 13], and magento-resistive memory and FeRAM both have similar expected high-density trends[13].

Simple extrapolation of memory behavior from these smaller sizes, however, may be flawed because an artificially small memory size can result in frequent page faults. As a result, writes to memory become increasingly correlated with page replacements rather than the actual write behavior of benchmarks, artificially inflating the write counts. This behavior is especially pronounced for write prevention techniques that rely on hamming distance information, such as [3], when page-faults cause cell-writes from a page-fill that overwrites old data from an evicted page. When sized properly, this intra-process cross-page writing would occur far less frequently, or maybe even not at all if the benchmark fits within memory.

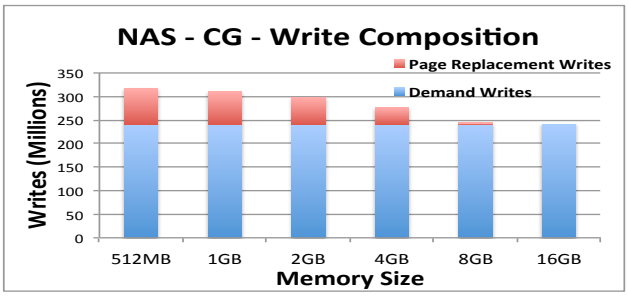

Figure 2 : Write composition breakdown into page replacements and demand writes

As shown in Figure 2, we demonstrate that for the *cg* benchmark from the NAS suite [9], write behavior increase by as much as 25% when modeling a 512MB Memory in lieu of 16GB. For multi-workload experiments this effect will be even more pronounced as applications are now competing for limited resources, increasing or creating inter-process conflict pressure that may not exist at all at the full memory size.

Table 1 : Benchmark Characteristics Sorted by Benchmark Suite (Lexical) and Main Memory Writes (Ascending)

		Instructions Billion	Main Memory Writes Million	Main Memory Reads Million	Memory Foot Print	Max Write 16GB Memory	Execution Time sec
BioBench	phylip	1769	**0.4**	4.5	3.1MB	6391	435
	clustalw	1418	**6**	3	3.4MB	17381	231
	tiger	601	**770**	18818	1.5GB	3712580	1171
	hmmer	1193	**4446**	216	58.9MB	977190	343
NAS - class D (8 threads)	cg	4312	**408**	23357	8.5GB	59	3714
	mg	54	**542**	270	16.1GB	326	65502
	lu	550	**3428**	4411	8.9GB	465156	10082
	is	1052	**11134**	28494	33.1GB	262551	79298
Parsec - simlarge (8 threads)	blackscholes	6	**0.7**	0.23	4.5MB	1093	0.7
	freqmine	36	**5**	51	172.2MB	77565	9.1
	bodytrack	14	**7**	10	15.5MB	25365	0.8
	x264	14	**30**	34	37MB	41099	0.8
	ferret	27	**65**	180	67.9MB	22057	2.2
SPEC	soplex.pds-50	12	**95**	190	151MB	11938	5.8
	perlbench - diffmail	381	**328**	94	331MB	2725369	81.6
	h264.ref_baseline	513	**427**	460	35MB	231970	9
	mcf - ref	372	**856**	27181	1.6GB	5368756	689.4
	bzip2 - source	429	**1040**	2221	857MB	487793	118.2
	gcc - g23	190	**3150**	2285	1.1GB	363794	117.6

Another contributing factor when using reduced memory extrapolation is simply the volume of cells being written to. As discussed previously, the operating system induces natural wear-leveling, even in the absence of other wear leveling mechanisms. Over time, memory traffic behavior will be spread out over all of memory, and a larger memory has more cells to distribute this traffic across. Therefore, the MTTF estimate for a modeled memory that is ¼ the projected size of an actual non-volatile main memory system will be at least 4 times lower than it should be, due just to this increase in cells.

C. Benchmark Execution Region

A well-established problem in the architecture community is the long simulation times when fully simulating benchmark execution. Many cycle-accurate simulation frameworks operate in the 10s of thousands of instructions per second (KIPS) to several 100s of KIPS range. However, instruction traces (as shown in Table 1) for full execution of benchmarks is measured in billions, or even trillions, of instructions. Even when assuming an aggressive 500 KIPS simulator, 1 trillion instructions still takes 555 hours (~23 days) to simulate.

While many acceleration techniques already exist to address this problem, such as SimPoints[16] and sampling methodologies, many researchers still apply a naïve 'skip X instructions and execute Y instructions' approach for evaluation. Since this is already a well-documented problem, we will only briefly present numbers demonstrating the errors associated with this kind of naïve extrapolation, and focus on what concerns need to be addressed when applying a more intelligent acceleration technique, Simple Random Sampling (SRS), to main-memory wear endurance simulation.

IV. Simulation Methodology

A. Simulation Environment

To demonstrate the impact of naive extrapolation, simulations were run to show the errors associated with each of the pitfalls discussed in Section III. Experimentation is done via trace driven simulation through Pin [12]. For NAS [9] and PARSEC [8], 8-threaded traces are captured on a machine with 8 Intel Xeon X5450 running at 3 GHz. For BioBench [11] and SPEC [10], single-threaded traces are captured on the same hardware. For all benchmarks, an on-chip 64K/4-way associative L1 and 512K/8-way associative private L2 cache per core/thread is modeled to filter the reference stream before issuance to main memory. The cache line size is set to 64B for both L1 and L2 caches.

To provide better understanding of the benchmark behaviors in the following discussion, a characterization of the benchmarks based on common metrics of interest affecting wear endurance is provided in Table 1. For these evaluations, results are collected from a single run of each benchmark, executed to completion modeling a 16GB main memory. By choosing 16GB, most benchmarks (except NAS's *mg* and *is*) do not exceed the main memory size, thus minimizing the OS paging effects from this characterization. It is worth noting that the values collected by our environment are consistent with previous characterization of these benchmark suites. The metrics of interest are as follows. The first data column holds the instruction count for the collected traces. Next, the main memory write count of the second column and read count of the third column is an indication of how many requests reach main memory after being filtered through the on-chip memory hierarchy. The main memory footprint is an indication of how many unique memory locations are touched during the execution. Next, the max-write is the count of the largest number of writes that map to a single cache-line sized memory block. As a result, this metric is often used in the wear endurance community for projecting MTTF. Even when ignoring the inter-process OS wear leveling effects discussed in this paper, we do not advocate using this methodology since MTTF should be based on a percent of memory failing, rather than first cell failure. These numbers, for all benchmarks except *mg* and *is,* can be viewed as equivalent to an unlimited memory simulation since their footprints do not exceed the 16GB main memory size.

Finally, native non-instrumented execution times for the benchmarks were collected using **/usr/bin/time** and are shown in last column. These values can be used to get a rough approximation of per-iteration execution time in calculating MTTF in terms of years. Benchmarks that ran for less than 30 seconds are executed a hundred times to obtain their average execution time because measurement errors due to daemons and other non-execution related effects may be too large of a contribution to the reported time and skew the estimate.

B. *Operating System Modeling*

The operating system tracks page usage of a process by maintaining an approximation of working set, via the reference bits. On each reference to a physical page, the processor sets a bit (the PG_referenced bit (PG) in Linux) in the page table entry. Periodically, the OS will update the memory usage of the process based on the pages with set reference bits and then clear these bits. This process also provides the OS with LRU information based on the granularity of the periods.

Beyond the reference set of an application, the OS must also maintain memory in distinct sets based on usage. Dirty memory is written in the background to the swap file on disk, enabling it to be more quickly reclaimed for other usage. Many pages are already holding copies of data that exist as files on disk, both the process binaries as well as memory mapped files and other constructs. This data is part of the file cache and can be reused across processes, as it is a clean copy of what is on disk.

Under normal client loads, most memory is dedicated to the file cache, while a server load will have a larger share specific to the single application. In either scenario, some pages will be clean and therefore easy to repurpose. Too many clean pages will require a heavy disk load to keep up with writes. Too few pages and the OS will have to double page (write to disk before reading in new data) to repurpose memory. And finally, the OS will try to maintain a small number of free or zeroed pages that are available for immediate allocation. Every free page is effectively wasted memory, as that page is not holding useful data but rather reserved for future usage.

The simulator provides an approximation of the OS paging techniques, just as the simulator approximates the processor and memory systems. While the following mechanism is specifically based on Linux, it is similar to that used in all modern Operating Systems. As shown in Figure 3, the OS paging system consists of an active and an inactive list. A free list is also included for new virtual to physical mapping requests. When a page is requested for the first time, a new translation entry is created, which is put on the front of the active list (accesses when the page is not resident are treated similarly). The new additions to the active list from the free list

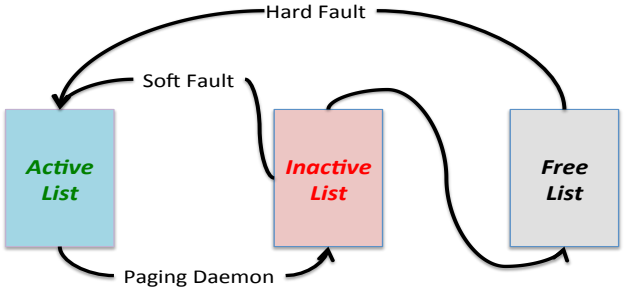

Figure 3 : Transition of physical pages among page list

are termed hard or major faults. When a page on the inactive list is accessed, it is transferred onto the top of the active list. This is termed a soft or minor fault, as there was no translation present, but the page resided in memory. During its time on the active list, if this page is accessed again, its PG bit is set (PG = 1). Every second the page daemon interrupts to rebalance the page lists. Equation 1 is commonly used in Linux to determine the amount of rebalancing to attempt, where nr_pages is a parameter usually set to 32, and nr_active and nr_inactive are the current size of their respective lists. The active list is traversed from bottom and any page with PG = 1 is put on top of the active list and its PG bit is reset. Pages with PG = 0 are transferred over to the top of the inactive list. The list traversal continues until a sufficient number of pages (Equation 1) have been moved onto the inactive list. Pages needed for new translations are evicted from the inactive list and put onto the free list [17].

$$\text{Pages to Move} = \frac{\text{nr\_pages} \times \text{nr\_active\_pages}}{(\text{nr\_inactive\_pages} + 1) \times 2}$$

Equation 1 : Number of pages to move from Active List to Inactive List

Furthermore, as a simulated L2 has been used to filter the memory access stream, the page usage information is based on operations going to memory, whereas a real system would update references on every memory operation including those hitting in the cache. The memory references obscured by the filter cache are proportional to the L2 cache size. Given the disparity between cache and system memory sizes, this effect should have minimal impact on the overall results.

A more significant effect would be to model the Operating System's file cache. Without modeling pages that may be reused between iterations, the simulator introduces additional writes to the system by filling pages with data that would already be cached in memory. Properly modeling this behavior would reduce the write load going to memory and therefore improve the lifetime.

V. EXPERIMENTS

The experiments presented here will demonstrate the degree of impact incorrect extrapolation methodologies can have on estimating lifetime of limited endurance memories. This section is divided into three subsections, covering the pitfalls from Section III: time extrapolation, memory size extrapolation and accelerated simulation. For all our results we pessimistically assume 100% bit changes for every main memory write.

A. *Lifetime Extrapolation*

When estimating lifetime from a single run, the basic assumption is that the rate of endurance degradation is constant over time, and therefore the results of a single execution can be used to project the degradation after several consecutive runs of a benchmark. However, due to the inter-process effects discussed in Section IV, this assumption is actually incorrect. To demonstrate this phenomenon, Figure 4 shows a plot of MTTF (worst case, assuming first failure causes failure) vs. number of consecutive executions for the cg benchmark from the NAS suite.

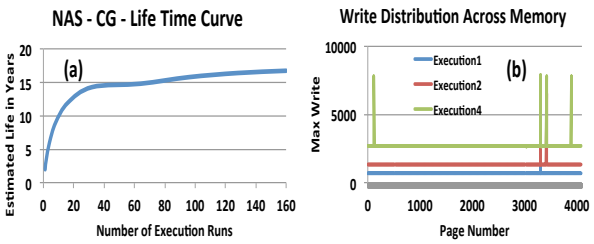

Figure 4 : NAS - CG – 512MB memory size (a) Lifetime Estimation vs. Number of Execution Runs (b) Physical Memory Write Distribution for 1, 2 and 4 Execution Runs

The most striking observation in Figure 4 (a), is that the estimate for MTTF nearly doubles when comparing the projection from a single run vs. two consecutive executions of cg. This is because cg has a non-uniform write profile across its memory footprint. In Figure 4 (b), only for visualization, we have grouped physical memory pages. However, our simulator models writes to every physical page at the granularity of cache line sized blocks within a page. Specifically, there is a spike (~6000 writes) in the write profile to a single page in memory, shown in Figure 4 (b), while all other pages have more uniform, lower write behavior (~700 writes). When run twice, the hot page has moved to another location and there are now two spikes of ~6700 writes. The max write count has only increased by 12%, but the execution time doubled. After 4 executions, the spikes are even more

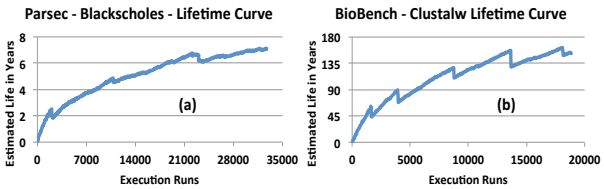

Figure 5 : Estimated Life Time vs. Number of Execution runs (a) BioBench – Clustalw (b) Parsec – Blackscholes

spread out. Hence, the wear does not increase at the rate a single execution run would have predicted.

Meanwhile, a naïve extrapolation from a single run, that assumes constant degradation, would have assumed the max write count to increase linearly with the number of execution runs. At a memory size of 512MB, there are approximately 128 thousand pages to select from, but only one is stressed per execution. Even when taking the birthday paradox into account, 128 consecutive executions have only a 0.405% chance of collision to the same stressed page between any two executions. Without collisions, the max written location after 128 executions would have ~95000 writes, yielding a lifetime estimate of 15.8 years, compared to the 2 year lifetime projected from a single run. This is supported by the empirical results in Figure 4, collected from simulation. In general, there are very few hot pages, which are largely responsible for the MTTF projection.

To demonstrate this, Figure 5 shows a plot of projected MTTF vs. number of consecutive executions for different benchmarks. Due to variations in execution times of individual benchmarks, the number of executions is not the same for all benchmarks.

The endurance degradation over multiple runs is clearly non-linear. The write distribution over a large number of execution runs tends to get uniformly distributed. The dips in the lifetime curve in Figure 5 happen due to "hot page" collisions. A "hot page" is a write-heavy page. When a hot physical page gets re-mapped to a write-heavy virtual address, a collision is said to occur. This implies that rate of memory wear out temporarily jumps up resulting in a dip in estimated lifetime. But gradually as number of execution runs becomes large, due to law of large numbers, the effect of such collisions on estimated lifetime is significantly reduced.

B. Memory Size Extrapolation

When observing results from the first run of a benchmark, the estimated lifetime between different memory sizes can lead to wildly different conclusions. A counter intuitive example is that of SPEC CPU2006 – MCF benchmark. The MCF benchmark has a high memory footprint and its memory access pattern causes frequent page replacements at lower memory

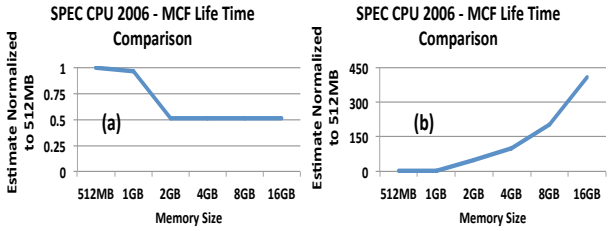

Figure 6 : SPEC CPU 2006 – MCF Estimated Lifetime vs. Memory Size (a) Estimate after First execution (b) Estimate at Perfect Wear Leveling

sizes. This causes frequent changes in virtual to physical address translations causing the writes to spread out over the memory. Whereas at higher memory sizes, the number of replacements are reduced and writes are not naturally distributed. As we see from Figure 6, the lifetime estimates after the first execution lead to a conclusion that lower memory size yields better lifetime. But as the number of executions increase, the wear leveling across the physical memory begins to take effect. Also it should be noted that, lifetime estimates do not increase linearly with increase in available memory size. As noted earlier in case of NAS – CG, reduction in writes due to page replacements at higher memory sizes leads to better lifetime estimates.

C. Accelerated Simulation

Even the simulation of a single run can be prohibitively expensive for certain benchmarks. We propose simple random sampling (SRS) to reduce execution run time of the first run itself. The Line Write Profile (LWP) is the write profile of the benchmark, which captures all of the write accesses (demand writes) to any physical main memory page. The LWP is a write counter which, keeps count of the number of writes for every cache line sized block within a single page. The profile assumes significance over a very large number of simulation runs, when even spreading of writes across the memory will create a similar profile on every physical memory page. For a

cache line size of 64B and page size of 4KB, we have 64 cache lines within a physical page. We observed that the LWP from the 10% sampled execution and the full run are identical. The only difference is in the actual write count. Scaling the 10% LWP appropriately results in the same LWP as the full run. The max write count from the LWP indicates maximum memory wear out. Since the LWP is cumulative sum of writes for every physical memory page, averaging the LWP by the total number of physical pages allows us to calculate the average memory wear per execution run. We perform SRS on the memory accesses rather than instructions, as wear out is determined by writes, and not the number of executed instructions. Memory operations tend to happen in bursts during phases and hence sampling must be performed on number of memory accesses and not instructions. In Figure 7 for the Bio - Bench – Tiger workload we observe similarity in the profile pattern for a full simulation and sampled simulation. As seen in Figure 8, the LWP obtained when simulating only the first 10 Billion or 100 Billion instructions for Tiger results in very different profiles, which look very dissimilar to the original profile in Figure 7. Needless to say, this must result in wildly varying estimates for memory lifetime. We collected the sampled LWP from 10, 15, 20, 25 percent sampled execution runs and observed less than 2% error across all lines in the LWP for BioBench – Tiger (Figure 9) and less than 5% error in write counts per line in the LWP for all benchmarks. On the contrary if we simulate only a fraction of the instructions from a benchmark, we can see a stark contrast in the LWP obtained between two such execution runs.

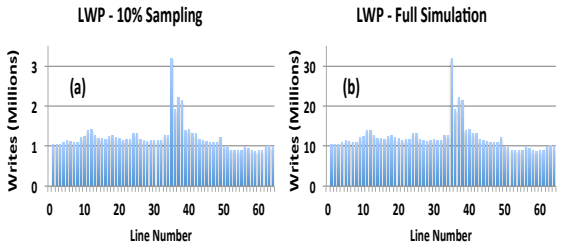

Figure 7 : BioBench – Tiger LWP Memory Access Sampling (a) 10% Sampling (b) Full Simulation

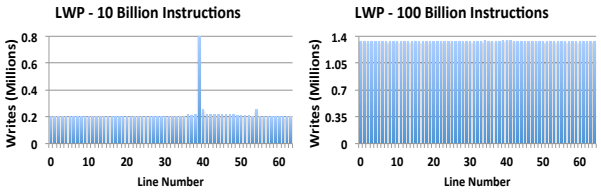

Figure 8 : BioBench Tiger LWP (Instruction Sampling)

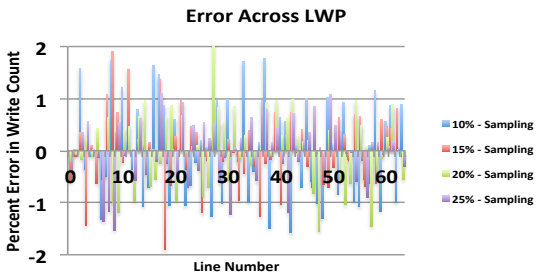

Figure 9 : Percent error in write count across all lines in LWP of Tiger for 10, 15, 20, 25% Sampling

D. Heuristic

Although sampling can be used to considerably speedup simulations, it is still impractical to simulate thousands or even millions of execution runs to simulate for entire memory lifetime. A quicker methodology is needed which estimates point of memory failure accurately. We have already demonstrated in Figure 4, that over multiple execution runs, the write distribution across the physical memory tends to become uniformly distributed. Under such assumptions we develop an analytical model.

Let there be N pages in memory, each of which can be written W_{max} times. Let μ_1 be the average writes per page and σ_1 be the standard deviation, during one execution run. Let us focus on a generic physical address p. The physical address p will be associated to a logical address. During the next execution run, the physical address p will be associated with a different logical address. Due to the randomness of the OS paging mechanism we can assume that the new logical address will be chosen at random. After a large number of execution runs, the total writes for line p can be approximated with a Gaussian distribution using the *Central Limit Theorem* [15]. After k execution runs, the expected value of sum of writes (Sum_k) and the standard deviation (σ_k) to page p is:

$$\mathbf{Sum_k = k \cdot \mu_1} \quad (1)$$

$$\mathbf{\sigma_k = \sqrt{k} \cdot \sigma_1} \quad (2)$$

The probability that the page p fails after k execution runs:

$$\mathbf{P\{Page\ p\ fails\} = P\{Z > W_{max} - Sum_k\}} \quad (3)$$

where Z is a zero mean unit variance Gaussian random variable. The heuristic predicts the mean and standard deviation of the write distribution for every execution run and calculates the corresponding lifetime estimate. We observe in Figure 10, that this estimate is very close to the results from the actual execution runs. Thus not only does our heuristic follow the simulation results, but it also shows that over a large number of execution runs (or execution time) the OS paging mechanism is fairly random in nature and causes uniform write distribution across the physical memory.

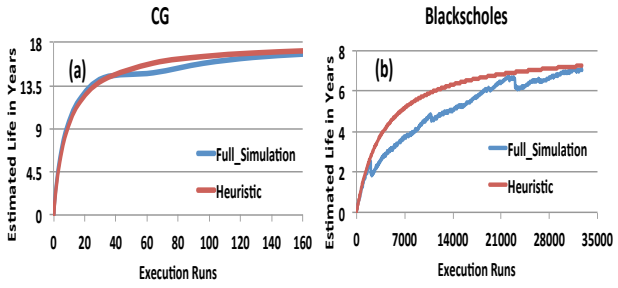

Figure 10 : Heuristic vs Full Simulation Lifetime Curve Comparison (a) CG (b) Blackscholes

VI. RESULTS FOR ESTIMATED LIFETIME

A. Heuristic Estimates for Lifetime at Wear Out

We simulate using our heuristic model to find the true lifetime of the memory. This true lifetime can be defined as the point at which memory cells reach their endurance limit, starting from a single run, until memory wears out.

Figure 11 shows maximum achievable lifetime for different memory sizes. The effect of operating system wear leveling is more pronounced for larger memory sizes. At smaller memory sizes, there is a possibility of wear out due to aggressive page replacements. By the time OS approaches uniform wear, a smaller memory size faces the danger of having already worn out enough to reach its maximum endurance. Whereas, for larger memory sizes, most of the benchmarks achieve close to perfect wear leveling and reach within 90 – 100% of perfect lifetime. It can be inferred that the paging mechanism does play a significant role in wear distribution for limited endurance memories. Even in the absence of any wear leveling techniques, the OS paging system causes near 100% wear leveling for most applications.

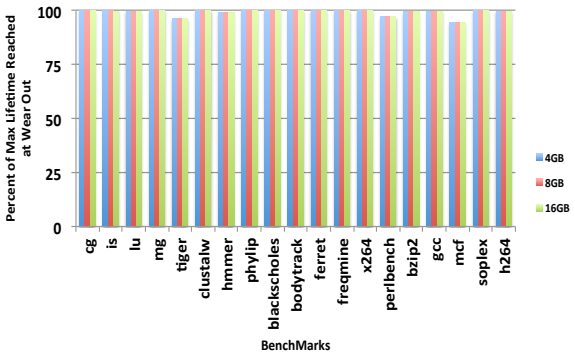

Figure 11 : Effect of Natural Wear Leveling by OS

B. Scope for Wear Leveling Beyond Perfect Page Leveling

The OS performs wear leveling at the page level by remapping physical pages over time. This causes distribution of writes across all pages in the memory. However, for certain benchmarks, the LWP is not uniform and there is still scope for further wear leveling at the intra page level. Lines within a page can be further re-mapped to attain perfect wear leveling at the intra–page level or the line level. Previous work by Qureshi et al. [14] explores this area in detail. Although it must be noted that any further wear leveling within a page is in addition to wear leveling across all pages in the memory.

VII. CONCLUSION

Wear leveling research will be increasingly critical as new types of write limited non-volatile memories emerge. Hence accurate lifetime estimates for limited endurance memories assumes greater importance. We have demonstrated the importance of modeling the operating system effects on wear leveling both within and between applications. The random nature of OS page allocations, results in a near uniform write distribution over the lifetime of the memory. As longer simulations for memory wear out are prohibitive we have also proposed a heuristic to derive faster lifetime estimates. We have also provided a heuristic that enables evaluation of memory endurance across its lifetime. This work provides researchers with faster and accurate simulation methodology for future wear leveling research.

VIII. REFERENCES

[1] Benjamin C. Lee, Engin Ipek, Onur Mutlu, Doug Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative", in ISCA' 09.

[2] Moinuddin K. Qureshi, John Karidis, Michelle Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, "Enhancing Lifetime and Security of PCM Based Main Memory with Start-Gap Wear Leveling", in ISCA' 09.

[3] Wei Xu, Jibang Liu, Tong Zhang, "Data Manipulation Techniques to Reduce Phase Change Memory Write Energy", in ISLPED'09.

[4] Gaurav Dhiman, Raid Ayoub, Tajana Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System", in DAC'09.

[5] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem and Daniel Mosse, "Increasing PCM Main Memory Lifetime".

[6] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, Michele M. Franceschini, "Practical and Secure PCM Systems by Online Detection of Malicious Write Streams".

[7] Nak Hee Seong, Dong Hyuk Woo, Hsien Hsin Lee, "Security Refresh: Protecting Phase Change Memory against Malicious Wear Out".

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li, "The Parsec Benchmark Suite : Characterization and Architectural Implications", in PACT'2008.

[9] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S.

Weeratunga, "The NAS Parallel Benchmarks," RNR Technical Report RNR-94-007, March 1994.

[10] SPEC Benchmarks http://www.spec.org/

[11] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A Benchmark Suite of Bioinformatics Applications", in ISPASS'05.

[12] Vijay Janapa Reddi, Alex Settle, and Daniel A. Connors, Robert S. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education," in WCAE'04

[13] Gerhard Müller, Nicolas Nagel, Cay-Uwe Pinnow, Thomas Röhr, "Emerging Non-Volatile Memory Technologies".

[14] Moinuddin K. Qureshi, Viji Srinivasan, Jude A. Rivers, "Scalable High Performance Main Memory System Using PCM Technology", ISCA '09.

[15] S. Ross, "*A First Course in Probability"*, Pearson Prentice Hall, 2006.

[16] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder, "Using SimPoint for Accurate and Efficient Simulation", International Conference on Measurement and Modeling of Computer Systems, June 2003.

[17] Mel. Gorman, "Understanding the Linux Virtual Memory Manager", Prentice Hall 2004