Using Emulation Software to Predict the Performance of Algorithms on NVRAM

Jana Traue, Jörg Nolte
Distributed Systems/Operating Systems Group
BTU Cottbus - Senftenberg
Cottbus, Germany
{jtraue,jon}@informatik.tu-cottbus.de

Philipp Engel, Reinhardt Karnapke
Distributed Systems/Operating Systems Group
BTU Cottbus - Senftenberg
Cottbus, Germany
{engelphi,karnapke}@tu-cottbus.de

ABSTRACT

Currently, new storage technologies which unite the latency and byte-addressability of DRAM with the persistence of disks are being developed. This non-volatile memory (NVRAM) may start a software revolution. Traditionally, software was developed for two levels of storage and NVRAM reduces the hierarchy to a single-level store. Current research projects are already exploring the potential of NVRAM, but they face a challenge when they want to evaluate the performance: The new hardware is not yet available.

In this paper, we discuss why benchmark results which are gained on existing DRAM are insufficient for a prediction of the performance on NVRAM. Either existing instructions have to be changed or new ones have to be introduced. We further show that the bochs emulator can be used to build systems which resemble NVRAM, to predict the NVRAM's consequences, and it even allows a comparison of algorithms for NVRAM.

Categories and Subject Descriptors

I.6.3 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Experimentation, Performance

Keywords

Emulation, Non-volatile memory

1. INTRODUCTION

In the early beginnings, computer systems contained only a single level of storage. As the processors became faster, the storage technology did not keep pace and memory access was going to slow the more powerful systems down. In order to speed up memory access, the technology had to lose one of its important features: long-term persistence. As a consequence, today's systems have several layers of volatile memory and persistent storage devices. Up to now, these memory types do not have much in common. On the one hand, volatile memory, DRAM, is byte-addressable and has a read/write latency of approximately 50ns. Storage devices, on the other hand, are orders of magnitude slower and transfer data with a block-oriented granularity.

Table 1: Access latency of different memory technologies on a 2 GHz machine (derived from [3, 12])

Memory Type	Read	Write
Cache	1-14 cycles	1-14 cycles
DRAM	100 cycles	100 cycles
NVRAM (PCM)	200 cycles	2000 cycles

This situation is about to change with the advent of non-volatile memory (NVRAM), like PCM and Memristors [12]. These upcoming technologies are promising candidates for a reunion of main memory and storage. Aside from the fact that NVRAM is going to be byte-addressable and persistent, very little is yet known about the technology, especially regarding its performance and limited write endurance. But researchers already came up with several use cases and possible advantages of the new hardware and want to show that these ideas are applicable as soon as possible. Without real hardware on their hands, emulators can be used to rebuild the hardware's functionality. The problem is that emulators are only useful for functionality checks, not for benchmarks or other statements concerning the performance of algorithms. One idea, which is used by some projects, is to run the applications on today's DRAM and report the number of processor cycles that were used. In contrast to emulation, benchmarks on DRAM give a first impression of the performance on NVRAM, but we claim that it is insufficient.

In this paper, we first discuss the architecture of future systems with NVRAM and underline that benchmarks on today's hardware cannot be used for performance predictions of algorithms for NVRAM. Afterwards, we introduce a new methodology for performance predictions and show how we extended the bochs emulator for this purpose.

2. SUPPORT FOR CACHE CONTROL

Caches outperform even today's DRAM by a factor of 100. Because NVRAM is expected to be slower than DRAM, especially when data is written (see Table 1), future systems will still contain volatile caches and buffering writes will be-

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254796

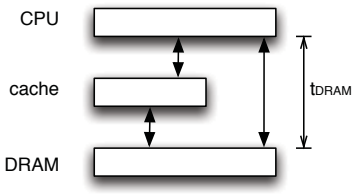

Figure 1: Simplified system with one core, one cache level and a DRAM module.

come essential. Upon an unexpected power outage, the data that was not written back from a volatile buffer to the persistent store is lost. In order to avoid such a loss, the programmers need to control when data is written back. Up to now, the contents of both cache and main memory are lost when the power runs out and persistent data exists only in files. As a result, it was seldom necessary for a programmer to know when modified data was written back to main memory. In this section, we summarize today's cache control mechanisms and discuss their suitability in systems with NVRAM.

2.1 Today's Systems

A simplified computer architecture that illustrates caching contains only a single core with one cache level and a memory module (see Fig. 1). When data is accessed for the first time, it is loaded from main memory, stored in the cache, and further operations on the data may use the cached version. In comparison to main memory, the cache is relatively small and, eventually, becomes full. At this point, a new load operation has to evict an existing entry from the cache in order to store the new one. The cache replacement policy may choose an unmodified entry from the cache and that means the version in the main memory is consistent with the cached one. Consequently, the cache entry can be overwritten. If instead a dirty entry is evicted, the modification has to be written back to main memory first. This is normally the time when modifications manifest in main memory, assuming that the software does not perform any cache control.
Is it possible for a programmer to predict this point in time? The answer requires a detailed understanding of the cache architecture, the replacement policy, a history and prediction of events, like interrupts, and is therefore hard to give.

But some situations, like communication with devices, force a programmer to control the cache write back. Today, Intel and AMD systems provide the following instructions [7, 1]:

wbinvd (write back invalidate) writes dirty data back to main memory and invalidates the whole cache.

clflush (cache line flush) writes a single cache line back to main memory and invalidates it in the cache.

Write back invalidate distinguishes between internal and external caches and, according to the manual, the write back for external caches is only triggered. Hence, the processor does not wait for data in external caches to be written back. In addition, `wbinvd` is a privileged instruction, what means that it cannot be used in user mode.

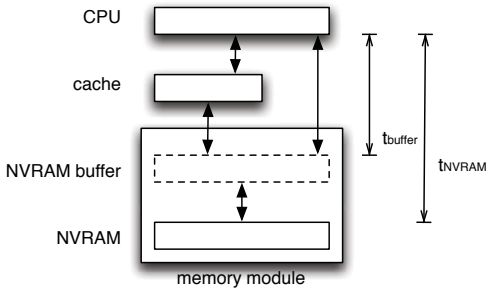

Figure 2: Simple system with NVRAM. The memory module contains a buffer in order to hide the access latency and perform wear leveling.

When a cache flush is issued, the programmer wants all data that is cached at this point to be flushed. Reordering the store operation could result in the cache line flush being carried out before the cache line is manipulated. This situation can be avoided by embracing the flush by *fence* operations. Such a fence operation assures that all preceding loads and stores become globally observable. Thereafter, the data can be written back.

2.2 Future Systems with NVRAM

In future systems, NVRAM could replace DRAM completely and the overall architecture would remain the same. But NVRAM writes are an order of magnitude slower when compared to DRAM (see Table 1) and the system's performance would be significantly degraded. Such a system is unlikely to become a success and volatile buffers have to be used in order to gain acceptable latency. Therefore, the memory module will not only contain non-volatile memory, but also feature a DRAM-like buffer, as shown in Fig. 2. Another benefit of such a buffer is that it allows memory access to be intercepted. Since the NVRAM cells' endurance is limited, wear leveling is essential. Although wear leveling could be performed by software, it requires such a detailed understanding of the underlying hardware that it is best performed by the hardware itself. With the help of the buffer, wear leveling can be made transparent.

Because the buffer is made of volatile memory, its contents are lost when the power runs out and write operations which have not yet been written to the actual NVRAM are lost. As with the other caches, persistent algorithms need to control the buffer's write backs. One option is to use the existing cache control instructions, but they have to be extended.

2.2.1 The wbinvd Instruction

Similar the usage of today's `fsync` and `msync` system calls, programmers may want to make changes to large data regions persistent in systems with NVRAM and, therefore, write all dirty cache lines back to the persistent store. The current implementation of `wbinvd` would signal a write back command to the buffer, but would continue immediately, because it does not wait for external caches to write their data back. If the operation's semantics are changed, the performance will suffer. In a volatile system with n_{clines} in its cache and access latency of t_{DRAM_w} for DRAM writes and t_{DRAM_r} for reads, a worst cases assessment of the cost of `wbinvd` is possible. Assuming that all cache lines are dirty,

all of them have to be written back. Because they are also invalidated, the subsequent operations have to fill the cache again. In summary, the costs are:

$$t_{wbinvd} = (t_{DRAM_w} * n_{clines}) + (t_{DRAM_r} * n_{clines})$$
$$t_{DRAM_w} = t_{DRAM_r}$$
$$t_{wbinvd} = 2 * t_{DRAM} * n_{clines}$$

With PCM as NVRAM technology, the read latency doubles and the write latency increases by a factor of 20. An extended `wbinvd` would therefore lead to a cost of:

$$\begin{aligned} t_{wbinvd} &= (t_{NVRAM_w} * n_{clines}) + (t_{NVRAM_r} * n_{clines}) \\ &= (20 * t_{DRAM_w} * n_{clines}) + (2 * t_{DRAM_r} * n_{clines}) \\ &= 22 * t_{DRAM} * n_{clines} \end{aligned}$$

As a result, the performance of `wbinvd` would degrade by a factor of 11. In a system with 15 MB cache, a cache line with a size of 64 Byte running with 2 GHz, the resulting speed would drop from about 12 ms to 132 ms. Whether the resulting penalty is acceptable depends on the frequency of `wbinvd` instructions. In the case that a new instruction `wbNVM` can be introduced, we would extend it to cover the NVRAM, but omit the cache line invalidation, which is not required for persistence. Without the invalidation, it is not necessary to fill the cache after the write back and this instructions' slowdown would be reduced to 10. We come back to this point in section 3.

2.2.2 The clflush Instruction

The other instructions that we discuss are `clflush` and the corresponding `sfence` operation(s). Removing an element from a linked list usually requires only a few pointers to be changed. Flushing the whole cache would make these modifications persistent, but it would also cause unnecessary flushes of the remaining data in the cache. Instead of issuing the `wbinvd` instruction, the modified cache lines could be tracked and flushed individually. Without fences, two subsequently issued `clflush` instructions might be carried out in a different order.

When two concurrent processes modify a shared list, one might perform modifications and mark them as complete afterwards. The complete marker should not be set before the modifications are complete. Issuing a fence operation avoids this situation because it waits for all previous stores to be globally visible. Unfortunately, the term globally visible is not very precise. With a cache coherence protocol being present, the fence might be limited to a store buffer flush on the issuing core. Writes that reach the cache are made observable by the cache coherence protocol. Because most modern multi-core processors are cache coherent, we assume the fence to end at the cache level. Similarly, the operations that follow afterwards should not be carried out before the marker is set, because the other process might be stalled in the meantime. Therefore, a second fence has to be used after the marker is set. The pseudo-code in Listing 1 summarizes the sequence.

With non-volatile memory, the cache line flush cost increases because the write is issued to the NVRAM and the subsequent read access the NVRAM, too. Similar to the `wbinvd`, the fence operation is eleven times slower. If it is frequently used, it seems desirable to keep its weak ordering, so that multiple flushes can be issued and a fence is required to assure their completion. In that case, the flush cost would stay the same, but the fence changes. The new fence has to wait until all previous flushes are written back from the volatile memory buffer and therefore its cost increase depends on the number of stores that is has to wait for.

Listing 1: Application of sequences of flush and fence operations

```
1 ... // modify data
2 fence(); //**
3 flush(); //* manifest changes
4 fence(); //**
5 ... // mark operation as complete
6 fence(); //**
7 flush(); //* manifest marker
8 fence(); //**
```

The consequences of our observations are twofold: First, either the semantics of existing instructions need to be extended for NVRAM or new instructions have to be added. Second, benchmarks with algorithms that operate on persistent data and use today's DRAM and instructions, are unlikely to produce usable predictions for NVRAM. We need emulation software to address these issues.

3. EMULATION SUPPORT FOR NVRAM

In the previous section, we have shown that the performance of algorithms on today's hardware is not comparable to NVRAM. Still, we want to perform experiments with NVRAM. Without the hardware being on the market, emulation software can be used.

3.1 Benefits of Emulation

Although emulation software does not allow cycle accurate simulation of all parts of a system, it is usable for experiments with new hardware. With respect to NVRAM, the following features are useful: an emulation of the persistent memory, the ability to add new instructions, and the opportunity to count selected events.

3.1.1 Rebuilding Persistence

In order to add NVRAM support to an emulator, it is possible to use files to preserve the main memory contents during runs. By editing the files manually, it is possible to simulate hardware failures, like incomplete operations or bit flips.

3.1.2 Adding new Instructions

As we stated earlier, the write back invalidate instruction could be extended or a new instruction, write back non-volatile memory without invalidation (`wbNVM`), could be introduced. In contrast to real hardware, an emulator allows to introduce new instructions. On the downside, it is not possible to emulate the whole semantics of these instructions, for example if the software does not emulate caches. Still, adding new instructions can be useful, for example to determine the frequency of their usage.

3.1.3 Predicting Performance

In order to predict the benefit of new instructions or the impact of extending existing ones, it is necessary to find out

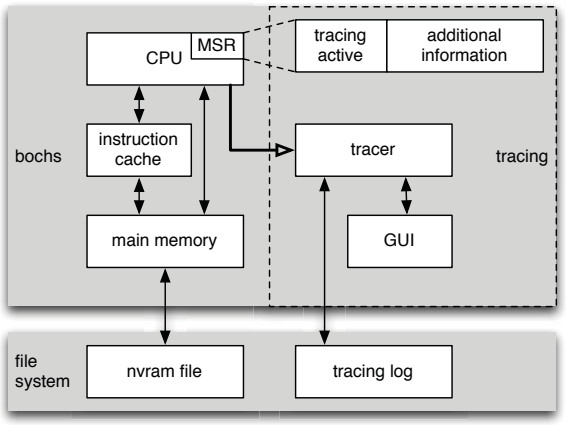

Figure 3: Architecture of the modified bochs with non-volatile memory and tracing support.

how often they are used in existing programs. With the help of an emulator, these instructions can be counted. In addition, other events which are also commonly not traceable on real hardware, like memory accesses, could be counted. The collected numbers allow a comparison of algorithms and a performance prediction.

3.2 Adding NVRAM Support to bochs

Bochs [2] is an open source emulator for the x86 platform. We have been using it for a couple of years as part of one of our lectures where students implement their own operating system and even added new features to the emulator. We added two features to bochs: persistent memory and an event tracer.

Persistent main memory is resembled by replacing the existing allocation of anonymous memory with a memory mapped file. In order to validate our implementation, we used the Linux hibernate mechanism. Normally, hibernate writes a copy of (nearly) the entire RAM to disk. We implemented a block driver that allows to declare a fraction of the NVRAM as disk. Upon reboot, Linux fills the memory with the previously stored data. This scheme is very inefficient because it creates persistent copies of already persistent data. Nonetheless, it allowed us to perform first experiments with the emulation platform.

The event tracer was implemented with the help of bochs' instrumentation interface. The instrumentation interface uses callbacks to intercept selected events, for example the occurrence of an interrupt or a specific instruction that was executed. We used the existing callbacks for tracing flush operations and added new ones for fence instructions. Our results can either be written to a text file or visualized by a GUI at run-time of the emulation. We can start the tracing when the simulation starts and control it by writing to a selected Model Specific Register. The resulting architecture is shown in Fig. 3.

Figure 4 shows an example output for the boot process of a Linux system with SMP support. Even when the system reached the log-in screen and no further input was given, the number of fence instructions continued to rise. We tracked the cause and found out that, because multiple cores are active, the Linux scheduler uses fence operations for synchro-

TRACING RESULTS

CLFLUSH	0
WBINVD	6
MFENCE	89672
SFENCE	33
LFENCE	69236

Figure 4: Tracing GUI for bochs which reports the number of selected events.

nization. If the semantics of the existing fence operations were extended to cover the volatile buffer of NVRAM memory modules, all of the scheduler's fence operations would slow down. These first results indicate that the semantics of existing operations should not be changed and new versions of fence operations should be added to the instruction set of upcoming processors.

4. RELATED WORK

The project that is most closely related to ours is presented by Zhu et.al. in [13] and explains how NVRAM emulation was added to bochs. Although not explicitly stated, the article indicates that the authors use memory mapped files to preserve the contents of the main memory. In that case, the emulation of the persistence of NVRAM is similar to our approach. Additionally, the authors simulate NVRAM's access latency by delaying read and write operations for a configurable number of milliseconds. Furthermore, they trace the number of writes to individual memory blocks in order to collect data for future wear leveling schemes. The main difference to our project is that we do not assume the memory hierarchy which bochs emulates to be realistic. Without modifications of its original source code, bochs has an instruction cache, but no other cache. Counting memory accesses would therefore include loads and stores that would be satisfied by the cache on real hardware. In contrast to the number of loads and stores that hit main memory, the number of executed instructions is already realistic and we rely on it.

Further projects develop software for NVRAM, like file systems (SCMFS [11] and PRIMS [6]), or persistent data structures (CDDS [9]). These projects report results from benchmarks that they run on DRAM without considering the different access latency of NVRAM or the introduction of new instructions.

Mnemosyne [10] is a persistent heap for user space applications. In order to make data persistent, the authors use the write-combine buffer and fence operations. They emulate NVRAM latency by using a ram-disk and delaying fence and flush operations for a configurable amount of cycles. Since they run on traditional hardware, they are not able to extend the processor's instruction set and cannot introduce new features.

Similar to Mnemosyne, NV-Heaps [4] also provides user-level persistent heaps. It relies on a modified hardware which provides 8 byte atomic writes and epoch barriers from the BPFS project [5]. For performance evaluations, the NVRAM's latency is simulated with Pin [8] and the results are combined with performance counter values from a real processor. Therefore, their predictions are limited to the events that modern CPUs are able to report, like cache hit

rates, but cannot count the number of selected instructions which were performed.

5. CONCLUSIONS AND OUTLOOK

In this paper, we have shown that NVRAM cannot simply replace DRAM in traditional systems, because the cache control is limited. Persistent data cannot be used without extending the semantics of existing instructions or adding new ones. As a consequence, benchmark results which are collected on DRAM do not predict the performance of the same algorithms on NVRAM adequately. We have discussed the features of emulation software that make it suitable for emulating NVRAM and its consequences. Furthermore, we reported first results that we collected with a modified bochs emulator.

In the future, we plan to use our emulator with existing NVRAM projects, like Mnemosyne. We plan to use Mnemosyne in our NVRAM emulator without the project's emulated PCM and compare our performance predictions to the ones made by its authors. In addition, we are currently working on transactional mechanisms for persistent data, plan to compare their performance and experiment with their robustness. We will enhance our performance predictions by considering even more events, like cache hits, which we gain from performance counters on today's hardware or by adding cache emulation to bochs.

Acknowledgments

We thank Intel for supporting our work with a research grant as part of the NOVOS project. We would further like to thank Thomas Prescher and Werner Haas for their valuable input during the development of this project.

6. REFERENCES

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*.

[2] bochs homepage. http://bochs.sourceforge.net/.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 46(3):105–118, Mar. 2011.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[6] K. M. Greenan and E. L. Miller. Prims: making nvram suitable for extremely reliable storage. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, HotDep'07, Berkeley, CA, USA, 2007. USENIX Association.

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[9] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[10] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. *SIGARCH Comput. Archit. News*, 39(1):91–104, Mar. 2011.

[11] X. Wu and A. L. N. Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[12] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li. Emerging non-volatile memories: opportunities and challenges. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 325–334, New York, NY, USA, 2011. ACM.

[13] G. Zhu, K. Lu, and X. Li. Scm-bsim: A non-volatile memory simulator based on bochs. In W. E. Wong and T. Ma, editors, *Emerging Technologies for Information Systems, Computing, and Management*, 236, chapter 109. Springer New York, 2013.