Whole-System Persistence

Dushyanth Narayanan Orion Hodson
Microsoft Research, Cambridge
{dnarayan,ohodson}@microsoft.com

Abstract

Today's databases and key-value stores commonly keep all their data in main memory. A single server can have over 100 GB of memory, and a cluster of such servers can have 10s to 100s of TB. However, a storage back end is still required for recovery from failures. Recovery can last for minutes for a single server or hours for a whole cluster, causing heavy load on the back end. Non-volatile main memory (NVRAM) technologies can help by allowing near-instantaneous recovery of in-memory state. However, today's software does not support this well. Block-based approaches such as persistent buffer caches suffer from data duplication and block transfer overheads. Recently, user-level persistent heaps have been shown to have much better performance than these. However they require substantial application modification and still have significant runtime overheads.

This paper proposes whole-system persistence (WSP) as an alternative. WSP is aimed at systems where all memory is non-volatile. It transparently recovers an application's entire state, making a failure appear as a suspend/resume event. Runtime overheads are eliminated by using "flush on fail": transient state in processor registers and caches is flushed to NVRAM only on failure, using the residual energy from the system power supply. Our evaluation shows that this approach has 1.6–13 times better runtime performance than a persistent heap, and that flush-on-fail can complete safely within 2–35% of the residual energy window provided by standard power supplies.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Main memory

General Terms Design

Keywords NVRAM, persistence

1. Introduction

Databases and key-value stores today commonly keep their working sets entirely in main memory. In-memory operation gives high throughput and low latency by removing I/O bottlenecks and hence is used even for large datasets. Today's servers can hold hundreds of gigabytes of main memory, and larger datasets are partitioned across multiple such servers. These servers might be viewed as caches for a back-end storage layer [5, 9, 19] or as main-memory databases that integrate in-memory operation with back-end storage [18, 30, 34]. In both cases, their key feature is that they hold a copy of the entire data set in main memory.

Recovery is a concern for these large main-memory deployments. After a power outage or any failure that causes loss of in-memory state, the entire state must be read or reconstructed from the storage back end. Typically this involves reading a recent checkpoint of the state and then replaying a log of recent updates. Recovering several terabytes of state in this manner is slow and puts a heavy load on the back end. For example, a Facebook outage in September 2010 [13] caused the service to be unavailable for 2.5 hours while in-memory cache servers refreshed their state from servers. The problem is that while in-memory operation and scale-out now allow the service to sustain very high levels of load during normal operation, the recovery mechanisms are I/O bound and they are not provisioned to deal with such spikes in load.

What if servers could recover their in-memory state locally and near-instantaneously after a failure? This would significantly speed up recovery and reduce back end load, especially after a correlated failure such as a power outage. This is possible if the server state is stored in non-volatile memory (NVRAM). Of course, servers can also fail due to application software errors. In this case, we must rely on the server application to handle the error locally (e.g. by rolling back a recent update) and/or to refresh its state from the back end. However NVRAM offers significant benefits for transient crash failures, specifically power outages or UPS failures that are correlated across a large number of servers.

In this paper we argue that it is feasible to use NVRAM in servers for recovery from transient crash failures. We further argue that current models for using NVRAM have significant limitations. Persistent buffer caches [6] and RAMdisks duplicate state, doubling the memory footprint of an in-memory application. They also suffer from block transfer and system call overheads. NVRAM-based persistent heaps [7, 33] achieve better performance than these block-based approaches by allowing in-place, user-space updates of persistent state. However, they still have substantial runtime overheads and require non-trivial changes to application code.

We propose *whole-system persistence (WSP)* as an alternative. WSP relies on all system main memory being non-volatile. It presents the application with the simple abstraction that all state is recovered after a power failure. In other words, a power outage is converted into a "suspend/resume" event with heap, stack, and thread context state all being restored. This is done transparently and does not impose any runtime overheads on the application. The key idea behind WSP is *flush-on-fail*: transient state held in processor registers and cache lines is only flushed to NVRAM on a failure, using a small *residual energy window* provided by the system power supply (PSU). This is the period of time for which the power supply continues to maintain output DC voltages after signaling a power failure. The residual energy window is a result of the internal capacitance of the PSU. Flush-on-fail eliminates the runtime overhead of flushing cache lines on each update.

ASPLOS'12, March 3–7, 2012, London, England, UK.

This paper makes three contributions. First, we propose whole-system persistence as the best programming model for using NVRAM, and discuss its advantages, especially with respect to the persistent heap model. Second, we describe an efficient design and implementation of flush-on-fail: saving transient state to NVRAM on failure rather than during program execution. Third, we show both the performance advantages and feasibility of the flush-on-fail approach through an empirical evaluation. Our results show that flush-on-fail performs 13x better than a user-level persistent heap [33] for an update-intensive workload, and 6x better for a read-only workload. They also show that saving transient state takes less than 5 ms across a range of platforms. By comparison, measurements of standard PC power supplies show that they can supply power to the system for 10–300 ms after raising a power failure signal.

Section 2 provides some background for this work: the increasing importance of in-memory operation, and the feasibility of using NVRAM in servers. Section 3 describes the design considerations that led us to the WSP model, including performance considerations, programming models, and failure assumptions. Section 5 presents the experimental evaluation that validates our key claims. Section 6 discusses the implications of our results, shortcomings of NVRAM-based recovery, and directions for future work. Section 7 describes related work and Section 8 concludes with a brief summary of the key contributions of this paper.

2. Motivation and background

Main-memory servers The main motivation for this work is the increasing prevalence of main-memory operation for databases. Servers with 100s of gigabytes of DRAM are easily available, going up to 1 TB [10] at the high end. This means that many previously disk-based data sets can now reside entirely in memory, fundamentally changing the system design assumptions. This is driving main-memory designs for both traditional SQL databases [18, 25, 30] as well as the ubiquitous "no SQL" databases or key-value stores [5, 9, 19, 34]. The latter typically use a "share-nothing" architecture, which means that they can scale out to 100s or 1000s of main-memory servers. This means that they can potentially approach petabyte scale with all data stored in main memory.

Main-memory operation gives high throughput and low latency by removing I/O bottlenecks. However, large memories give rise to a new problem: recovery times. After a crash or reboot, a server must recover its state, typically from back end storage. This is limited by I/O bandwidth, which has not scaled with memory sizes. Reading 256 GB at 0.5 GB/s from a high-end storage array will take more than 8 min, even if all the storage resources were dedicated to that single recovering machine. Correlated failures such as rack-level power outages will cause 10s to 100s of servers to concurrently recover their state from a shared back end.

Ideally, servers would recover locally and near-instantaneously from such failures, without causing "recovery storms" on the back end. We briefly examine current approaches to surviving transient power failures and some trends that motivate the use of NVRAM.

Power failures and NVRAMs Power outages are particularly relevant because (without NVRAM) it is not possible today to recover in-memory state locally after a power failure; and because power failures can be correlated across a large number of machines. This means that power outages are a problematic case for recovery load, causing large correlated spikes in load on the recovery subsystem.

Hardware solutions for power outages today mainly consist of *uninterrupted power supplies (UPS)*. These use large lead-acid batteries to maintain power to the entire system for up to several hours after a power failure. An UPS allows the system to continue operation until the battery is almost completely discharged. UPSes are bulky, adding to data center space requirements, and environmentally unfriendly. Additionally, UPS failures can lead to hundreds of machines suffering power outages simultaneously. Alternative designs, such as a "distributed UPS" [15], avoid some of these problems by including a small lead-acid battery in each server. However, this still retains the space, cost, environmental problems, and unreliability of lead-acid batteries.

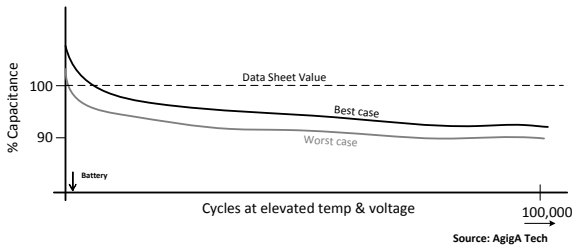


Figure 1. Effect of charge-discharge cycles on ultracapacitors

Battery-backed NVRAMs such as those used in RAID controllers typically augment DRAM (or sometimes SRAM) with rechargeable Lithium-ion batteries. When power fails the batteries maintain the memory contents while their charge lasts, up to a few hours. These batteries require frequent monitoring and replacement. Additionally they can only sustain a few hundred charge/discharge cycles before their performance degrades significantly. These disadvantages mean that battery-backed NVRAMs have been restricted to niche uses such as RAID controllers.

In the long term, *storage class memories (SCMs)* promise large capacities, low idle power consumptions, and non-volatility. Hence they seem ideal for in-memory server applications. However, they do not seem likely to replace DRAM in the near term (2–5 years). The most promising SCM candidate — phase-change memory (PCM) — requires additional additional hardware support such as fine-grained wear leveling [28] to be usable as a main memory alternative. This has only recently been proposed by the research community and it will be several years before PCM becomes commercially available as main memory. Additionally, PCM is expected to be significantly slower than DRAM, especially for writes [21] and hence will become competitive with DRAM only when the capacity benefits outweigh this disadvantage.

Battery-free NVDIMMs offer a practical and performant NVRAM option today. They are based on commodity components: DRAM, ultracapacitors and NAND flash integrated into a single module. The amount of NAND flash is equal to the amount of DRAM. The flash is not visible to the host system; it exists only as a backup and is not read or written during normal operation. When signaled by the host, the NVDIMM saves or restores the DRAM contents to/from the flash. Thus the NVDIMM gives the illusion of a non-volatile memory to the host system. The host is only responsible for flushing transient state to the NVDIMM over the memory bus and signaling the start of a save operation. After this, the NVDIMM's contents are preserved even if the system PSU and host lose power.

At least two recently developed products — AgigaRAM [1] and ArxCis-NV [32] — uses this approach to make NVDIMMs that are drop-in replacements for standard (volatile) DIMMs. The DRAM, flash, and controllers are all integrated onto the NVDIMM, which fits in a standard DDR2 or DDR3 DIMM slot. The ultracapacitor module charges from the system's 12 V DC power supply, and discharges to provide power to the NVDIMM when the system power fails. Unlike Lithium-ion batteries, ultracapacitors can tolerate hundreds of thousands of charge/discharge cycles with only a 10% loss in usable capacitance (Figure 1). Although they store much less to-

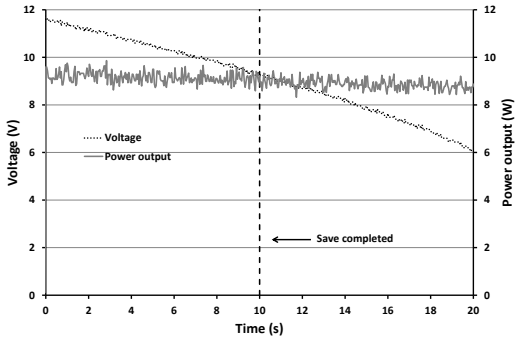


Figure 2. Voltage and power draw on ultracapacitors during NVDIMM save

tal charge than batteries, they can discharge it at a much higher rate to power the saving of DRAM contents to flash. Figure 2 shows the voltage delay and power output for a 1 GB NVDIMM. The save time is under 10 s for NVDIMMs of up to 8 GB and the ultracapacitor is able to supply power to the NVDIMM for at least twice as long[1].

Using flash-based NVDIMMs is not the same as saving system state ("hibernating") to a flash-based SSD. The latter requires the OS to suspend processes and devices, and to write out the state to a file on the SSD. This adds a long and variable latency, during which the entire system must be kept powered up. Additionally, the entire state must be written through a single shared memory bus and I/O channel. NVDIMMs by contrast save their contents off the critical path; there is no interaction with the host system during the save process, and the ultracapacitor backups ensure that the save does not rely on system power. Additionally, NVDIMMs can be saved/restored in parallel, since they do not use any shared resources.

This approach creates main-memory NVRAMs based entirely on cheap commodity components: DRAM, NAND flash, and ultracapacitors. Ultracapacitors (or supercapacitors) have achieved large economies of scale in recent years due to demand in the automotive industry, and costs are below $0.01/F and $2.85/kJ [14]. For comparison, the Agiga NVDIMMs require 5–50 F of capacitance and 0.1–1 kJ of energy, depending on size.

Thus the availability of commodity component based NVDIMMs today, and projected trends for storage class memories, make it likely that large main memory NVRAMs will be a viable option for servers. In the rest of this paper we assume servers where all main memory is NVRAM made up of NVDIMMs, i.e., non-volatile memory with DRAM performance.

3. Design rationale

This section gives the detailed design rationale for WSP. First, we distinguish failures that are recoverable using NVRAM from those that are not. Next we examine different programming models for using NVRAM. Finally we examine the performance implications of flushing transient state in processor caches on each update. We use these comparisons to motivate the two key features of WSP: to recover all application state rather than selected subsets of it; and to flush transient state on a failure rather than during execution.

[1] The internal voltage required by the NVDIMM is only 3.3 V, and hence a power input of 6 V is still usable.

3.1 Failure models

NVRAM is useful for recovery from crash failures, such as power outages. It cannot be used to recover from errors that corrupt state, such as software bugs. Other, software-based mechanisms exist for failures such as kernel panics [11] or application component failures [4]. Generic mechanisms, e.g. rolling back state on failure using checkpoints or software transactions, might also help in the case of software bugs.

These mechanisms work equally whether memory is volatile or non-volatile. This observation favors a design where recovery from crash failures using NVRAM is decoupled from other types of failure recovery. For the purposes of this paper we use "persistence" to mean survival of state after a crash failure rather than any other kind of failure. By decoupling persistence from other forms of recovery, it should be possible to support a wide range of programming models and recovery mechanisms, while at the same time providing resilience to crash failures.

In our model, NVRAM is the first but not the last resort for recovery after a crash failure. Recovery from a back-end storage layer such as a file system or database will always be necessary in some cases, e.g., if an entire server fails. The in-memory server is a cache, but one with a high refresh cost. Our aim is to reduce this cost by enabling local recovery in the case of power outages.

3.2 Models for persistence

There is a variety of ways in which applications can persist and recover state after a failure. We can broadly categorize them as:

1. *Block-based:* Applications propagate updates from the in-memory representation to a file, block device, or database. Typically the in-memory objects are first converted to a more suitable representation such as database records or serialized objects. On recovery, these are read back and deserialized to recreate the in-memory state.
2. *Persistent heaps:* Applications use a transactional API to allocate, deallocate, and modify persistent objects. On recovery, the application retrieves a special "root" object from which it can reach the others.
3. *Whole-system*: The application uses only in-memory objects, and does not distinguish between persistent and volatile objects. On recovery, all state is restored transparently.

In any of these cases, the state to be persisted can be stored in NVRAM, or on a disk or other back-end storage.

Block-based persistence on local NVRAM could be implemented as a persistent buffer cache [6], an NVRAM-based file system [8], or simply as a non-volatile RAMdisk. Regardless of the implementation, block-based persistence suffers from three disadvantages. First, it doubles the application's memory footprint by storing data in two locations. Second, the application must convert data between the two representations on each update and during recovery. Third, the application pays the overhead of block transfers and system calls to transfer data to the NVRAM.

NV-RAM based persistent heaps, or *NV-heaps* [7, 33] avoid these problems. Persistent objects are stored in NVRAM and mapped directly into the application's address space, and updated in-place. A transactional mechanism ensures that the persistent heap is always recovered to a consistent state after a failure. NV-Heaps thus combine three key features. First, they use *selective persistence*: the persistent heap is recovered after a failure but not volatile heap objects, stack objects, or thread contexts. Second, they use *flush-on-commit* of updates and transactional logs to NVRAM to ensure that updates are not lost after a power failure. Third, they use *software transactional memory (STM)* for isolation across threads and consistency of the persistent heap. Selective persistence is problematic for legacy applications; flush-on-commit causes sig-

nificant runtime overheads; and STM, while potentially useful, is not used by all applications.

Whole-system persistence (WSP) relies on all application memory being mapped to NVRAM. With NVDIMMs, we believe this is feasible simply by replacing all server memory with NVDIMMs. Thus with WSP only transient state (processor registers and cache contents) needs to be captured when a failure occurs; we address this through a flush-on-fail mechanism that writes this state to memory when a failure occurs.

We claim that the best use of NVRAM is for whole-system persistence (WSP). This can then be combined with a block-based model for the back end storage, e.g., applications can periodically checkpoint their state to a file. This allows instantaneous local recovery of state after a power failure, with more expensive recovery from the back end for more severe failures. With WSP and flush-on-fail, we eliminate the first two disadvantages of NV-heaps and make the third (use of STM) optional. WSP can be used transparently with single-threaded, lock-based, or STM-based applications. In the rest of this section we compare WSP with NV-heaps in more detail.

Selective vs. whole-system persistence Designating only certain objects as persistent opens the door to "dangling references": persistent objects that refer to volatile ones. After a crash/recovery, such references become unsafe, since the volatile objects have disappeared.

Such dangling references are hard to avoid when porting a large legacy application and even when writing an application from scratch. They include not just pointers from persistent to volatile objects but also indirect references, e.g., array indexes or hash table keys where the array or hash table is part of the volatile state. They also include handles to OS objects. The application code must ensure that such references are never stored in a persistent object. This makes transparent persistence — re-using existing data structures as persistent ones — difficult.

Legacy applications are particularly problematic, since their state needs to be cleanly partitioned between persistent and volatile. If the original application was not written with such a separation in mind, this can involve substantial rewriting, assuming that source code is available. Any code that might obtain a reference to a persistent object must be checked to ensure that it does not then create an unsafe reference. This includes third-party libraries, for which source is often unavailable.

Even if all source code is available, aliasing in C/C++ means that dangling references cannot always be found through static analysis. Since it is impractical to check all code by hand, there is always the risk of following a unsafe pointer. Although careful restrictions on persistent object types can reduce the probability of "programmers getting it wrong" [7], selective persistence still adds to the challenges of writing (or porting) programs correctly.

The "dangling reference" problem is avoided in models that use transitive persistence [3, 26]: objects are dynamically made persistent whenever a reference is created to them from a persistent object. However this requires a garbage-collected language such as Java [3] or SML [26] and is not suitable for the many server applications that are written in C or C++.

Flush-on-commit vs. flush-on-fail Flush-on-commit refers to the fact that persistent heap updates must be propagated to the NVRAM before the update is considered persistent. Transaction logs must be similarly written to the NVRAM rather than remain in processor caches.

Transaction logs can be either redo or undo logs. We evaluated an existing NV-Heap implementation, Mnemosyne [33] which uses redo logs; in addition it also uses STM for concurrency control. As a result, each write to the persistent heap is recorded in the transaction's write set at write time; generates a redo log record which is written to NVRAM at commit time; and requires a cache line flush at log truncation time to ensure that updates are not lost. In addition reads must be instrumented to check the writeset for local uncommitted writes. Finally, using STM for concurrency control adds additional overheads in the form of conflict detection at commit time.

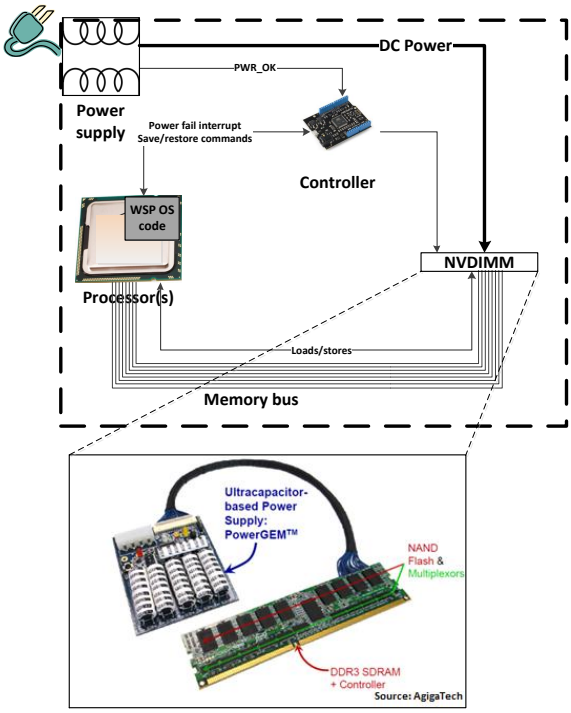


Figure 3. WSP hardware prototype

Not all the above operations are necessary for persistence; in fact Mnemosyne has significant overheads even for a read-dominated workload due to the instrumentation of reads. To remove these overheads, we implemented a minimal NV-Heap, which provides persistence (i.e. crash consistency) but not isolation (concurrency control). It uses an undo log rather than a redo log. Undo log records are written efficiently to a torn-bit raw log using non-temporal stores, as in Mnemosyne. For single-threaded operation, this performs significantly better than Mnemosyne; however, flushing of updates and log records still impose a high overhead.

The high overheads of runtime flushing motivated our "flush-on-fail" approach. Section 5.1 describes our experimental evaluation of NV-heaps in detail.

Summary Persistent heaps are an inappropriate mechanism for using NVRAM. They impose a problematic programming model and require substantial changes to legacy applications. They offer transactional consistency, which might be useful for recovery from a specific class of software failures. However, if this is desired, exactly the same recovery semantics can be enabled, with better performance, by using a non-persistent transactional heap combined with WSP. Alternatively, legacy applications can use WSP to recover from power and hardware failures, with no change in their behavior after software failures.

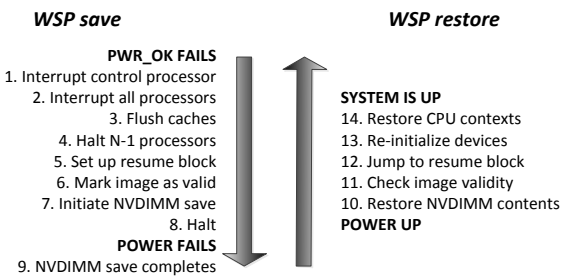


Figure 4. Save and restore steps for WSP

4. Design and Implementation

Implementing WSP requires four components. A hardware *power monitor* signals the system when a power failure is detected. A software *save routine* saves and flushes all transient state to RAM, and halts all processors, before power is actually lost. A corresponding *recovery routine* restores the saved state when power is next restored. Finally, one or more hardware *NVDIMMs* are required to implement the NVRAM functionality. Figure 3 shows a schematic of our hardware prototype. The software components are implemented on the Windows Server 2008 R2 OS running on a 64-bit PC platform.

Power monitor The power monitor is implemented using a NetDuino programmable microcontroller. It monitors the ATX power supply's PWR_OK signal [16]; the power supply drops this signal as soon as it detects an input power failure. The microcontroller then triggers an interrupt on one of the host processors via a serial line.

Save and restore routines The save routine is invoked by the interrupt handler corresponding to the serial line connected to the power monitor. The key functionality of the save routine is to implement "flush-on-fail". One of the processors — the control processor — co-ordinates the flush process. It issues an "inter-processor interrupt" —- essentially a task that executes at high priority — to all the other processors. In parallel, all processors save their processor context to memory and flush their caches using the x86 `wbinvd` (Write Back and Invalidate Cache) instruction. All processors other than the control processor then halt. The control processor waits for all the tasks to complete, sets up a resume context, writes and flushes a "valid" marker to memory, signals the power failure to the NVDIMM(s), and halts. The valid marker is cleared on system startup and after a successful resume; it ensures that any failure during the save step is correctly detected.

The restore process is the inverse of the save routine, and is invoked on the next boot-up after a power failure. A modified boot loader signals the NVDIMMs to restore their saved contents. When the restore is complete it checks the valid image marker, and jumps to a resume context that is set up in a well-known location. This code in turn restores the saved contexts on other processors, and resumes normal scheduling. Figure 4 shows the save and restore steps.

Device restart The save/restore routines ensure that application and OS state (i.e. processor contexts and memory) are retained across a power failure. This means that, after a restore, the in-memory state of device drivers will be inconsistent with that of the devices, since the latter have been power-cycled. Also, kernel and/or user threads might be blocked waiting for a device operation to complete. To restore the system to a fully working state, devices and device drivers must be re-initialized, and device I/Os that were in flight at the time of failure canceled and either failed or retried.

There are multiple ways to do device restart. We have implemented and evaluated a simple strawman approach, which is to save all device state on failure using the existing support for putting the system in a sleep (S3) power state. In other words, we simply use the system's ACPI suspend/resume functionality to save and restore system and device context to memory, while relying on the NVDIMM support to retain the in-memory saved state. This approach is simple and transparent but can require significant amounts of additional time on the save path. Putting devices in a sleep state usually involves allowing all outstanding I/O to complete, which could take 100s of milliseconds or more on a heavily loaded and slow device such as a disk. Section 5, we show that the latency of using ACPI suspend significantly exceeds the residual energy window on our test systems, thus making this approach infeasible without adding a large amount of additional capacitance.

A better approach is to avoid any additional work on the save path, but instead to clean up device state on the restore path. We have investigated several ways of doing this, e.g., using plug-and-play support to "reboot" the device stack during restore. Although conceptually straightforward, this is complex to implement in a large OS such as Windows, and cannot handle all devices: e.g., legacy non plug-and-play devices, or the disk device holding the paging file which cannot be "unplugged".

A more practical approach to device restart might be to *virtualize* the devices, e.g. by using a virtual machine (VM) hypervisor. Virtualized server environments are common and increasingly so. In this approach, a fresh instance of the entire "host OS" and its physical device stack is booted after a failure. Individual VMs have their state restored from NVRAM, and the hypervisor transparently retries or fails outstanding operations on the virtualized devices. We are currently looking at implementing such I/O replay support in a VM hypervisor such as Hyper-V.

NVDIMMs The save and restore routines rely on main memory contents being preserved by the memory, i.e., the NVRAM. Our design uses AgigaRAM NVDIMMs [1], which use their own ultracapacitor charge to save their contents to on-board flash on a power failure. Thus it is sufficient to initiate the NVDIMM save within the residual energy window; the save will be completed without system power using energy from the ultracapacitors.

In our prototype, the microcontroller communicates with Agiga NVDIMM(s) over an I^2C bus. It translates commands sent from the host processor over the serial line to NVDIMM commands, i.e. "save" and "restore". The current version of the AgigaRAM NVDIMM requires the DRAM module to be put into "self-refresh" mode before the save or the restore operation can begin. In the case of "restore", the host processor also has to bring the memory out of self-refresh, and re-initialize the memory parameters. These steps require firmware (BIOS) support not provided by standard BIOS implementations (AgigaRAM is targeted at embedded systems with custom firmware). We are currently working on modifying a standard BIOS to support this.

5. Evaluation

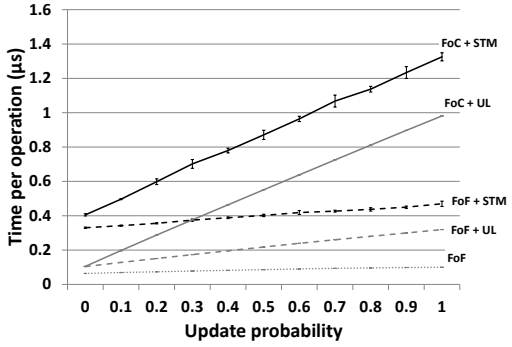
In this section we present experiments that answer three questions. First, what is the performance benefit of WSP's "flush-on-fail" approach over the "flush-on-commit" approach of NV-heaps, i.e., what is the performance gained from moving cache flushes out of the performance path? Second, how long is the residual energy window on typical PC power supplies? Third, how long is the "flush-on-fail" save time?

The testbed for our performance experiments was a 2.13 GHz 2-socket Intel C5528 system with 4 cores per socket and 2 hyper-

Configuration	Updates/s	
Mnemosyne	2160	(77)
WSP	5274	(139)

Means of 5 runs, with standard deviations shown in brackets.

Table 1. Update throughput for OpenLDAP



Each point is the mean of 10 runs, with min-max error bars.

Figure 5. Hash table microbenchmark performance

threads per core. The testbed has 48 GB of DDR3 DRAM (1333 MT/s). For the measurements of residual energy and save time, we measured this platform as well as a lower-powered AMD platform. This had a single 6-core AMD 4180 processor and 8 GB of DDR3 DRAM.

5.1 Performance

In our first experiment we compare the performance of an NV-heap implementation, Mnemosyne [33], with the WSP approach. We use the OpenLDAP benchmark reported by Volos et. al [33] with the same configuration parameters, on our hardware testbed running Linux 2.6.35. The benchmark inserts 100,000 randomly generated entries into an empty directory. The workload is processed by an OpenLDAP server that uses an AVL tree stored in the Mnemosyne NV-heap, rather than the default Berkeley DB, as the persistent store.

Mnemosyne ensures durability and consistency of the AVL tree by wrapping updates in transactions and logging updates in an efficient in-memory log structure. This log is written directly to memory, bypassing the processor caches, using non-temporal stores. We compared this with a WSP "flush-on-fail" version in which all transactional instrumentation and logging was disabled, i.e., a normal in-memory AVL tree. We did not undo the other modifications made to OpenLDAP to avoid dangling references; although these modifications are unnecessary with WSP, they do not add much performance overhead.

Table 1 shows the results of these experiments in terms of update throughput. This is a single-threaded, closed-loop experiment and hence operation latency is simply the inverse of throughput. We see that the WSP version is 2.4x faster than the Mnemosyne version. The overheads of Mnemosyne are partially due to the persistent log and flushing of updates; and partially due to other STM-related overheads such as tracking readsets, instrumenting reads, etc. These are not required for persistence, and since OpenLDAP is lock-based, they are not needed for concurrency control either.

To separate out these sources of overhead, we evaluated and compared five configurations using a simple hash table benchmark. The configurations are:

- *FoC + STM* (Flush-on-commit with STM): this is the default Mnemosyne configuration.
- *FoC + UL* (Flush-on-commit with undo logging): this uses flush-on-commit with an undo log (Section 3), and does not use STM.
- *FoF + STM* (Flush-on-fail with STM): uses the default Intel STM library, which instruments and logs transactional reads and writes, but does not flush logs or updates from the processor caches.
- *FoF + UL* (Flush-on-fail with undo logging): Undo logging is enabled but log appends and data writes are in-cache; they are not flushed synchronously to memory.
- *FoF* (Flush-on-fail with no transactions or logging).

As the non-STM configurations do not provide concurrency control, we ran the benchmark single-threaded: our purpose is not to compare different concurrency control mechanisms, but to measure the overheads of logging and flushing. The undo log was implemented on a Windows platform and relies on the Phoenix compiler framework [23] to instrument writes to the persistent heap, whereas the STM-based configurations use the Intel STM compiler [17] on Linux. The *FoF* configuration works on both platforms; we found that the Phoenix-based version was 11–17% slower, and we report this slower performance.

Each benchmark run pre-populated an in-memory hash table with 100,000 entries, and then measured the performance of 1,000,000 random operations on it. Each operation was either a key lookup or an update (with equal numbers of inserts and deletes). Figure 5 shows the results as the average time per operation as a function of the proportion of updates in the workload.

We can observe several things. First, the *FoC + STM* configuration is 6–13x slower than *FoF*. The penalty increases linearly with the update ratio, reflecting the cost of flushing updates and log records. By using undo logs (*FoC + UL*), the STM overheads are removed. For a read-only workload the remaining overhead (60%) is that of creating a transactional context for each operation, which is significant for short operations. For a write-intensive workload, *FoC + UL* is still almost 10x slower than *FoF*. This shows that it is synchronous flushing to memory that dominates performance, rather than other transaction-related costs.

This is confirmed if we examine the *FoF + STM* and *FoF + UL* configurations. Although slower than *FoF*, they are much faster than the flush-on-commit configurations. We conclude that even if transactions are desired for concurrency control or error recovery, it is still better to use them in-cache with a flush-on-fail approach, than to combine transactions with persistence using flush-on-commit.

5.2 Residual energy

To evaluate the feasibility of flush-on-fail, we measured the residual energy window and the time to flush-on-fail (save transient state) on both the high-end (Intel) and the low-end (AMD) testbed. We first present the residual energy measurements and then the timing measurements.

To measure the residual energy window, we used a sampling oscilloscope to monitor the PWR_OK signal as well as the voltage levels on DC output lines from the ATX power supply, with a sampling frequency of 100 kHz. When the power supply detects an input power failure, it drops the PWR_OK signal. We measured the interval from this signal dropping to the first voltage drop seen on any of the power supply's output lines. Since the samples are noisy, we define an output voltage drop as any 250 μs interval where the output voltage drops below 95% of its nominal value.

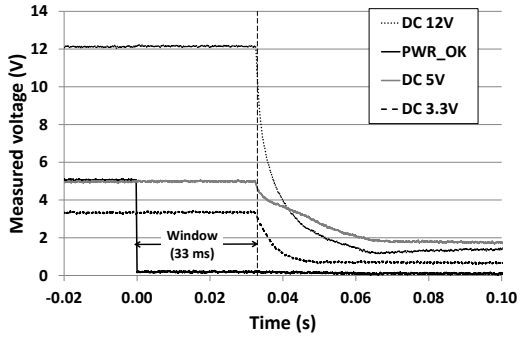
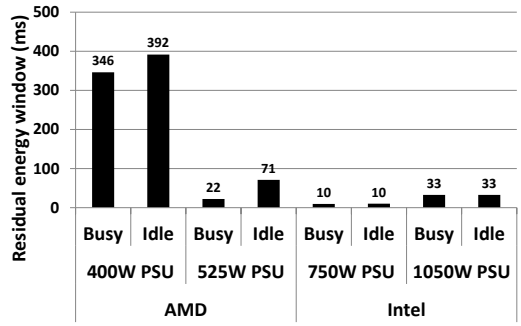


Figure 6. Residual energy window (Intel testbed)



Each value is the worst (lowest) observed of 3 runs.

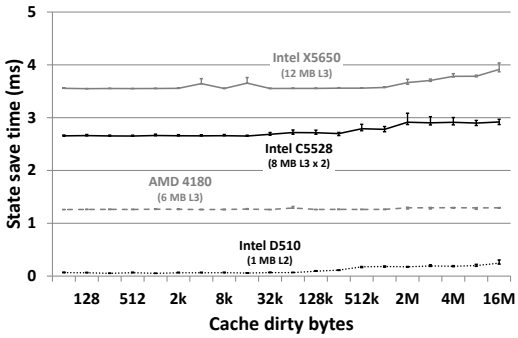
Figure 7. Residual energy windows across configurations

Figure 6 shows the voltage over time for the Intel platform, before and after the power failure signal. A CPU-intensive prime number calculator and a disk stress test were used on the high-end testbed to keep all processors and block devices busy. To ensure a worst-case configuration, all stress tests continue to run even after the power failure notification, thus keeping the system busier than with WSP. The residual energy window measured in this experiment was 33 ms.

These results used a 1050 W power supply. We repeated the experiment with a lower-rated 750 W power supply. We also tested the lower-power AMD system with two power supplies rated at 400 W and 525 W. For each configuration we measured both a worst case (all stress tests running) and a best case (idle). Figure 7 shows the residual energy window for each configuration. We see residual energy windows from 10–400 ms, depending on the power supply as well as the system being tested. This variation reflects the variation in internal capacitance across different power supplies.

5.3 State save time

Figure 8 shows the time to save all processor contexts and flush all caches on our two test platforms (Intel C5528 Nehalem and AMD 4180 Opteron) as well as two other Intel processors (Intel X5650 Xeon and D510 Atom). The cache invalidate/flush time dominates the total save time; using a simple benchmark we varied the amount



Values shown are means of 32 runs; error bars show min-max variation. Sizes in parentheses are those of the largest cache on each chip.

Figure 8. Context save and cache flush times.

	`wbinvd`	`clflush`	Theoretical best
2 x Intel C5528	2.8 ms	2.3 ms	0.79 ms
AMD 4180	1.3 ms	1.6 ms	0.65 ms

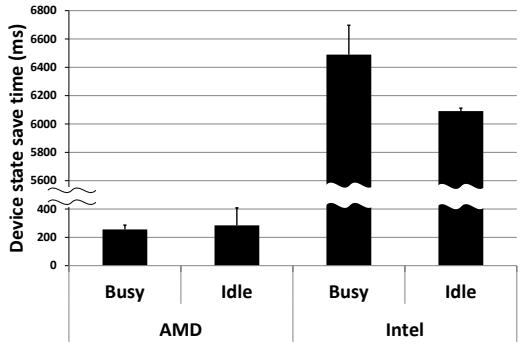
Table 2. Cache flush times using different instructions

of dirty data in the cache, up to the size of the largest cache on any of the processors. We see that save times are consistently under 5 ms in all cases, and under 3 ms for our two testbed platforms. Interestingly, there is little dependence on the number of dirty cache lines. We believe this is an artifact of the implementation of the `wbinvd` instruction. Flushing individual dirty lines (using the `clflush` instruction) is more efficient when there are few dirty cache lines; however it is not practical to track the location of dirty cache lines in software. Table 2 shows the worst-case (all cache lines dirty) flush times for `wbinvd` and `clflush`, as well as the theoretical best achievable based on measured memory bandwidths.

Thus the measured residual energy window on our two testbed platforms is 2.5–80 times larger than the save time, even in the worst case. This shows that it is feasible to use the residual energy to save processor contexts and flush caches. Further, the worst-case time to flush the caches is a function of processor cache size and memory bandwidth, and is easily characterized and measured for a given system. Similarly the residual energy window can be conservatively estimated as a function of capacitance and the maximum rated power draw of the system, and external capacitance added to provide any desired safety margin.

Device state save time The timing measurements in Section 5.3 are from a system that saves only processor state and does not save device state. We believe that the correct approach is to re-initialize devices and restart I/Os on the recovery path, rather than quiesce devices or save device state on the save path. However, it is interesting to ask the question: what is the additional latency of saving device state on the save path?

We measured the time taken by Windows to put all system devices into the D3 sleep state for both our testbeds and in both the busy and idle configurations. Figure 9 shows the results. We see that state save times are large and variable: comparable to the residual energy windows on the AMD testbed and much larger on the Intel testbed. Additionally, the "idle" save times are substantial, showing



Values shown are means of 5 runs; error bars show standard deviations. Note the break in the y-axis.

Figure 9. Device state save time

that even when there are no outstanding I/Os, the state save process is slow. We further investigated the high state save times for the Intel testbed and found that they are dominated by three devices: the GPU, the disk, and the network interface (NIC). The first is not required for a server configuration: in fact the only crucial devices for a server are the disk and the NIC. These device drivers could be modified to eliminate unnecessary timeouts and delays, and speed up the save process. However we believe that a better approach is to avoid executing any device driver code on the save path, and to use virtualization to transparently restart the device stack on restore.

5.4 Summary

We have shown that there are significant performance advantages to a "flush-on-fail" approach compared to "flush-on-commit" approach to data stored in NVRAM. Unlike flush-on-commit, flush-on-fail does not require STM or transactional logging; at the same time transactional applications also benefit from improved performance with flush-on-fail.

It is clear that processor contexts and cache contents can safely be saved well within the residual energy windows of standard PSUs. If desired, this residual energy can be explicitly provisioned by using a supercapacitor. For example, the state save on our test platform could be powered by a 0.5 F supercapacitor that costs less than US$2.

However, device suspend paths are slow and potentially unbounded since they rely on draining outstanding I/Os. It is not clear that we can rely on saving device state within a fixed time and energy budget. Instead, we must recover device state on the restore path, for example by using virtualization and replaying outstanding I/Os on the virtualized devices.

6. Discussion and Future Work

Our evaluation shows that flush-on-fail has better runtime performance than a flush-on-commit approach, and that standard power supplies provide a sufficient residual energy window to make this feasible. We believe that overall our results show promise for the use of main-memory NVRAM as the first resort recovery mechanism after a power failure. However there are still many open questions and directions for future work; here we discuss the most relevant ones.

Distributed applications Data center servers are often deployed today as part of a scalable distributed system. These systems already tolerate server failures, e.g. by using replication and/or failover techniques. On a failure, these systems must pay the cost of reading in-memory state from a checkpoint in the storage tier, or re-replicating the state from a live replica. For large memory servers, the cost of this state refresh can be substantial. With NVRAM and WSP, a server that suffers a short period of unavailability will have state that is stale but still mostly relevant, allowing a more efficient state refresh by only applying the recent updates that were missed during the failure. Some distributed applications maintain versioned state and/or update logs, allowing missed updates to be efficiently recovered. For other applications that are constructed as an "all or nothing" cache, this will require some added complexity.

Long outages So far we have assumed that power outages are relatively short (seconds to tens of seconds), and hence there is benefit in rapidly recovering state locally on the failed machine. However, with longer outages, the system might be forced to instantiate a new instance of the service on a different server, thus incurring the penalty of recovering from the back end. Does NVRAM change these tradeoffs, e.g., can we delay the re-instantiation of a server to allow NVRAM-based recovery? This is particularly relevant when using state machine replication [29]. If replication is done across uncorrelated servers (e.g. with independent sources of power), then the service can be kept available after a failure, with a new replica brought online in the background. This can potentially be delayed, to reduce the recovery costs in case the failed machine is able to recover with most of the state, i.e., lacking only the most recent updates.

SCM-based NVRAMs So far we have focused on DRAM-based, rather than SCM-based, NVRAMs. SCMs (storage class memories) such as phase-change memory and memristors have only recently emerged, and it is difficult to predict exactly what their performance will be, and whether/when they will become viable as server main memory. Broadly, they are expected to have access times comparable to, but slower than, DRAM. This will increase the overhead of flush-on-commit, especially for asymmetric memories such as phase-change memory, which is expected to be 10–100x slower than DRAM for writes but only 2x slower for reads [12]. Hence we expect that these slower memories will show an even larger performance advantage for flush-on-fail. SCMs are also expected to scale to much larger capacities than DRAM, potentially allowing multi-terabyte main memories. Note that the energy costs of flush-on-fail do not scale with main memory sizes, but only with the size of processor caches.

One potential disadvantage of SCMs such as phase-change memory is that they are expected to be more energy-intensive to write than DRAM. This might require additional capacitance in power supplies as a function of processor cache size, memory bandwidth, and the power consumption of writing the memory.

Hybrid systems With SCMs, there is also the potential for hybrid DRAM-SCM systems, with a small fast DRAM alongside a larger slower SCM. WSP can be used with such systems by replacing the DRAMs with NVDIMMs, with no impact on runtime performance. Regardless of persistence, however, hybrid systems raise an interesting performance challenge: automatically mapping objects and pages to either DRAM or SCM to maximize overall performance.

NVRAM failures Flush-on-fail adds an additional failure mode to the system, which is failure during the save process. A failed save will be detected on boot-up and trigger a normal recovery from the back end. However we believe such failures will be rare. The save routine is implemented at a very low level of the OS and required changing fewer than 200 lines of code. Thus software failures are unlikely. The internal capacitance of the PSU might unexpectedly fail to provide the required residual energy. It is straightforward and

cheap to provision the PSU with sufficient capacitance by using supercapacitors [22].

Process persistence So far we have assumed a model where both application and OS structures are restored after a power failure. An alternative is to save only application state process state and to restore it on a fresh instance of the OS. This still provides the same abstraction to the application as whole-system persistence: unlike the persistent heap model, thread contexts and stacks would be restored. It could also use the same fast save path for NVRAM. The recovery mechanism however would be different, since in effect application state will have to be separated out from OS state. Otherworld [11] is an implementation of process persistence for Linux. In Windows, application processes have dependencies on a variety of OS structures, making this approach complex. However, recent work on Drawbridge [27] has shown that most of these dependencies can be encapsulated into a "library OS" that can be included in the application's state, leaving a narrow and easily restartable interface to the OS kernel.

Future work Our immediate goal is to extend our prototype into a fully implemented and deployable system. This involves added firmware support for saving and restoring NVDIMM state, and hypervisor support for per-VM persistence. In the longer term, we intend to investigate failure and recovery tradeoffs for different scenarios: e.g., what are the costs/benefits of adding capacitance to a system compared to more frequent recovery from the back end.

7. Related Work

Persistence models Whole-system persistence is very similar to the orthogonal global persistence model of KeyKOS [20]. KeyKOS implemented global persistence through periodic checkpoints to disk and required extensive OS support. WSP is aimed at NVRAM-based systems and is based on flushing transient state to NVRAM on failure rather than during runtime, and operates with little or on OS modification. The persistent heap model has also been widely used in different language environments including PS-ALGOL [2], Java [3], and SML [26]. Section 3.2 provides in more detail our rationale for using whole-system or global persistence rather than heap persistence.

SCM based architectures Recent developments in SCMs have led to new designs for file systems and "on-disk" data structures optimized for these technologies. BPFS [8] is an in-kernel file system that leverages the byte-addressability of SCMs to provide fast file operations. Similarly the Rio buffer cache [6] preserves file buffer cache contents after a "warm reboot"; when used with NVRAM it means that file writes can be considered durable when they are written to the buffer cache. Both BPFS and Rio leverage non-volatile memory to speed up file writes. For an in-memory server, however, even such fast file operations cause significant runtime overheads as well as duplication of state. These overheads are reduced significantly in the persistent heap approach and eliminated in the WSP approach. CDDS B-Trees [31] are non-volatile B-Trees optimized for byte-addressable rather than block operation; they outperform traditional block-based B-Trees by 75–138%. NV-heaps allow use of a wider range of data structures such as hash tables, binary trees, and skip lists. WSP is transparent to applications and any in-memory data structures can be used.

Flash-based NVRAMs eNVy [35] proposed the use of SRAM backed by batteries and flash, to implement the abstraction of a byte-addressable non-volatile store on the main memory bus, i.e., a large NVRAM. The Agiga NVDIMMs are similar (with the use of ultracapacitors and DRAM rather than batteries and SRAM). A key difference is that in eNVy the SRAM was used to buffer only a small subset of the flash contents: with a random-access workload, the system would be bottlenecked on paging to and from the flash. NVDIMMs by contrast leverage dropping DRAM prices to store the entire contents in DRAM, with the flash layer being written/read only on failure/recovery.

Device recovery Our WSP prototype takes a simple approach to saving and recovering device state, which takes much longer than flushing processor caches and contexts. Ohmura et al. [24] propose an alternative approach: shadowing device registers in NVRAM by logging register writes on each device operation. This removes eliminates the "state save" at the cost of additional complexity in the device driver. A more device-independent approach is to re-initialize all device-related state, and replay device operations as necessary. Otherworld [11] reboots the entire operating system, transferring control to a new "crash kernel". Application state is maintained, and application threads that are in system calls have those calls aborted with a failure code; they can then retry the I/O. Although the aim of Otherworld is to recover from operating system crashes, this approach could also be used as a way to re-initialize device state after a WSP save/restore. Similarly, Drawbridge [27] allows an entire application, together with its OS "personality", to be serialized and restarted on a new kernel. This could be used after a WSP save/restore to extract application state safely, after which the OS can be restarted.

8. Conclusion

Main-memory servers are common in data centers today and will be increasingly used, driven by the need to avoid I/O bottlenecks, the availability of large-memory (up to 1 TB) DRAM systems, and the potential for even larger byte-addressable storage-class memories (SCMs). These servers introduce the problem of recovering a large amount of in-memory state from back-end storage after a failure. Non-volatile main memory (NVRAM) can alleviate this problem, but current software models for using NVRAM have limitations. In particular, user-level persistent heaps, while outperforming block-based NVRAM, still have high overheads compared to in-memory operation. They also require applications to be rewritten to a new API and memory model.

In this paper we argued for, and showed the feasibility of, an alternative approach: whole-system persistence. This allows legacy applications to run unmodified and with no overheads, by flushing transient state at failure time rather than during execution. We showed through experiments that this "flush-on-fail" can be powered using the residual energy from the system power supply, and that this gives a runtime performance benefit of 1.6–13x compared to a user-level persistent heap.

References

[1] AgigaTech. AGIGRAM (TM) Non-Volatile System. `http://www.agigatech.com/agigaram.php`, 2012.

[2] M. Atkinson, K. Chisholm, and P. Cockshott. PS-algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[3] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, Dec. 1996.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - A technique for cheap recovery. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, San Francisco, CA, Dec. 2004.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the ACM Symposium on Operating Systems Design and Implementation (SOSP)*, Lake George, NY, Nov. 2006.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Cambridge, MA, Oct. 1996.

[7] 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. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, CA, Mar. 2011.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[10] Dell. Dell poweredge r910 4u rack server. `http://www.dell.com/us/enterprise/p/poweredge-r910/pd?~ck=anav`, July 2011.

[11] A. Depoutovitch and M. Stumm. Otherworld: giving applications a chance to survive OS kernel crashes. In *Proceedings of the European conference on Computer systems (EuroSys)*, pages 181–194, Paris, France, Apr. 2010.

[12] E. Doller. Phase change memory and its impacts on memory hierarchy. `http://www.pdl.cmu.edu/SDI/2009/092309.html`.

[13] Facebook engineering notes: More details on today's outage. `http://www.facebook.com/note.php?note_id=431441338919`, Sept. 2010.

[14] Foresight. T2+2 (tm) market overview: Supercapacitors. `http://batteries.foresightst.com/resources/MarketOverviews/NET0007IO.pdf`, Dec. 2009.

[15] J. Hamilton. Open compute ups & power supply. `http://perspectives.mvdirona.com/2011/05/04/OpenComputeUPSPowerSupply.aspx`.

[16] Intel. Atx specification (version 2.2). `http://www.formfactors.org/developer/specs/atx2_2.pdf`, 2004.

[17] Intel. Intel C++ STM Compiler, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, Aug. 2010.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[19] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *Proceedings of the ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Big Sky, MT, Oct. 2009.

[20] C. R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 86–91, Sept. 1992.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Austin, TX, June 2009.

[22] Maxwell. Maxwell boostcap ultracapacitors. `http://www.maxwell.com/products/ultracapacitors`, Dec. 2011.

[23] Microsoft. Phoenix technical overview. `https://connect.microsoft.com/Phoenix/content/content.aspx?ContentID=4513`, July 2011.

[24] R. Ohmura, N. Yamasaki, and Y. Anzai. Device state recovery in non-volatile main memory systems. In *COMPSAC*. IEEE Computer Society, 2003.

[25] Oracle. Oracle TimesTen in-memory database overview. `http://www.oracle.com/technetwork/database/timesten/overview/timesten-imdb-086887.html`, July 2011.

[26] J. O'Toole, S. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 161–174, New York, NYA, Dec. 1993.

[27] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, Newport Beach, CA, Mar. 2011.

[28] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Austin, TX, June 2009.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, Dec. 1990.

[30] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1150–1160, Vienna, Austria, Sept. 2007.

[31] 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 USENIX Conference on File and Storage Technologies (FAST)*, pages 61–75, San Jose, CA, Feb. 2011.

[32] Viking Technology. ArxCis-NV (TM) Non-Volatile Memory Technology. `http://www.vikingtechnology.com/arxcis-nv`, Aug. 2012.

[33] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, CA, Mar. 2011.

[34] VoltDB. `http://voltdb.com/`, July 2011.

[35] M. Wu and W. Zwaenepoel. eNVy: A non-volatile main memory storage system. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, 1994.