Implementing Persistent File Cache on Android Operating System

Prakhar Panwaria, Prashant Saxena, Robin Paul Prakash
Computer Sciences Department
University of Wisconsin-Madison
{prakhar, prashant, rprakash}@cs.wisc.edu

Abstract

In conventional systems, memory is treated as a non-persistent storage. Because the contents of memory are volatile, changes to the data stored in memory are periodically written back to the disk. But some applications, which require higher reliability, explicitly flushes the file cache to disk at a much higher rate, thus affecting performance and energy consumption. This memory's perceived unreliability forces a trade-off between performance and reliability. In battery backed systems like mobile phones, the chances of unexpected power failures are much lesser when compared to conventional systems, which implies that data in memory is likely to be safe for a much longer duration. We leverage this idea to develop a model which modifies the file cache to behave as a persistent file cache, and hence increase system performance and power efficiency. Since data is kept in memory for a longer duration, it is more prone to corruption due to kernel bugs like buffer overflows. We propose two mechanisms - write protecting pages and use of transcendent memory, to address this issue. We also present analysis of the performance and power improvements of our system.

1 Introduction

In the modern storage hierarchy, I/O devices such as hard disk drives are considered to be reliable storage medium, whereas random-access memory (RAM), being volatile, is viewed as unsafe to store any data for a long duration. However, the access time of memory is significantly lesser than that of disks. Hence, to increase the performance, systems try to read bulk of data from the I/O device to the cache and then read it directly from the cache rather than reading it multiple times from disks. Similarly, data-writes to disk are also buffered in caches and are flushed to disk in larger blocks (to reduce disk seek time) and at larger intervals (to reduce number of disk accesses). Applications that desire higher reliability flush data to the disk explicitly at much higher rates, in order to minimize data loss in case of unexpected power failures or system crashes. Though this leads to better safeguarding of data, it increases the number of disk accesses, thereby decreasing the overall performance of the system.

It is intuitive to believe that if a better mechanism is provided to protect data in memory, then contents in memory can be treated as persistent and such reliability induced flushes to disk can be avoided. In battery backed systems like mobile phones, the chances of unexpected power failures are much lesser when compared to conventional systems, which implies that data in memory is likely to be safe for a much longer duration. A robust protection mechanism on top of persistent memory on such devices can deliver significant performance boost. We leverage this idea to develop a model for mobile platform operating system which modifies the file cache to behave as a persistent file cache, and hence increase system performance and power efficiency.

With these ideas in mind, our goal is to achieve performance of main memory with reliability of disk for mobile systems. The major parts of the design are:

1. Overriding flush requests from the application and managing interval at which data is

written to disk.

2. Implementation of mechanism to protect data in memory.

3. Securing contents of the memory in case of a system crash/reboot.

We present the design and implementation of our prototype on the Linux-based operating system from Google - Android [16]. Since handheld devices store permanent data in sdcards instead of spinning disks, we evaluate our system performance on sdcards only. However, for the sake of generality, we refer to disks as permanent media for storage in rest of the paper. Performance evaluation clearly shows merits of our approach. For write intensive workloads, we get performance gains up to 143% (in emulator) and 155% (in hardware) over standard Android kernel.

The rest of the paper is divided as follows. In Section 2, we discuss the research work that has been done around making file system more reliable, while improving its performance. In Section 3, we describe how we can make main memory behave as persistent storage. We also discuss mechanisms for in-memory data protection from system crashes and kernel bugs. In Section 4, we present the evaluation of our model, and show how modifying file cache parameters and avoiding flushing of data can enhance system performance and decrease power consumption. We present our conclusions in Section 5.

2 Related Work

In past, multiple attempts to model reliable main memory for persistent storage have been proposed [1,2,3,4]. Mnemosyne [1] deals with storage class persistent (SCM) memory. The research builds upon the fact that storage class memory provides data persistence inherently and hence proposes mechanisms to expose SCM directly to the application layer. There is a similar study around using NVRAM [2] (non-volatile RAM) along with client cache to improve file system reliability (data persists longer) and performance (less write traffic). Software Persistent Memory [3] talks about securing in-memory content by modifying data structure management schemes of particular language (C in their case).

Phoenix [4] attempts to make all permanent files reliable while in main memory. It keeps two versions of an in-memory file system. One of these versions is kept write-protected, the other version is unprotected and evolves from the write protected one via copy-on-write. At periodic check-points, the system write-protects the unprotected version and deletes obsolete pages in the original version.

Conquest [5] leverages the declining cost of persistent RAM technologies like battery-backed RAMs to implement a disk/persistent-RAM hybrid file system that holds small files in the battery backed persistent RAM and large files in disks. As a new file systems with separate data paths for in-core and on-disk storage was designed, changes were needed in the operating system as well. The performance tests using benchmarks shows a performance improvement of 43% to 97% over disk- based file systems.

None of the approaches to persistent storage described above are directly applicable to our model. Mnemosyne approach cannot be ported on Android based systems due to lack of SCM in mobile hardware. Instead, our approach focuses on providing standard disk based persistent memory. Software persistent memory has limited use in Android based systems since application memory management is mostly done transparently by Dalvik virtual machine. We propose modification at kernel level, which operates below the Dalvik virtual machine to provide support for persistent memory. Using hybrid file system entails major change in Android operating system, and hence is out of scope of our work.

What is more relevant is protecting the memory from operating system crashes and overwriting of application data through unauthorized accesses. Rio file cache [6,7] tries to address this issue by forcing all processes including the kernel to access memory through the virtual mem-

ory mechanism, thereby avoiding dangling kernel pointers overwriting application data in the event of a system crash. However, Rio File cache implementation as defined in [6] uses warm reboot, which writes file cache data to disk during reboot. Unfortunately, warm reboot relies on several specific hardware features, such as a reset button that does not erase memory, and is not applicable to most class of hardware used in handheld devices.

Techniques like Recovery Box [8], keeps backup copies to enable recovery of the system once it crashes. As the aim of our recovery mechanism is not to prevent applications from saving corrupted data, but to recover unsaved data in the event of a crash, we do not need a recovery mechanism based on checkpointing as we are able to save the entire memory contents before a reboot. Our work is inspired by SafeSync [7] approach, which is a software based technique that requires no hardware support. It writes dirty file cache data reliably to the disk during the last stage of a crash.

Our approach to memory protection primarily uses two mechanisms - write-protecting pages to avoid illegal accesses via corrupt pointers, and making a copy of dirty data in transcendent memory [9,10], which is not directly accessible by kernel. These two mechanisms provide the necessary protection to secure the contents of memory. By using zcache [11,12] to compress the pages, transcendent memory provides efficient storage of duplicate pages without causing too much memory overhead.

3 Design and implementation of a Reliable File cache

This section describes how we modified the file cache implementation in Android operating system to behave as a persistent file cache. We also describe mechanisms to protect in-memory data from system crashes and kernel bugs.

3.1 Persistent File Cache

As discussed in previous sections, reading and writing data off the memory is relatively fast when compared to I/O from disks. Hence, our aim is to retain the data in memory for longer duration and serve subsequent requests from cache. In conventional Android kernel, data migration from memory to disk can happen in two scenarios - a) Explicit transfer, where an application uses system calls to flush the data to the disk, or b) Implicit transfer, where OS periodically flushes the data from buffers to disk. In Android OS, implicit transfer happens every 30 seconds. Implicit transfer guards the user from losing data in case of power failure or system crash. In case of crash, only data between the implicit transfer checkpoints is lost. Since this window (30 seconds) is relatively small, very small amount of data is at risk of being lost.

In following subsections, we briefly define file synchronization mechanism adopted in Android Linux kernel and highlight major changes to achieve our goal of retaining data in memory.

3.1.1 File synchronization mechanism in Android Linux kernel

Both implicit and explicit transfers are performed using three system calls as defined below:

- *sync()* - Allows a process to flush all dirty pages to disk
- *fsync()* - Allows a process to flush all pages that belong to a specific open file to disk
- *fdatasync()* - Very similar to *fsync()*, but doesn't flush the inode block of the file

The service routine *sys_sync()* of the *sync()* system call invokes a series of auxiliary functions- *wakeup_bdflush()*, *sync_inodes()*, *sync_supers()* and *sync_filesystems()*.

wakeup_bdflush() starts a pdflush kernel thread, which flushes to disk all dirty pages contained in the page cache.The *sync_inodes()* function scans

the list of superblocks looking for dirty inodes to be flushed. The function scans the superblocks of all currently mounted filesystems; for each superblock containing dirty inodes it first invokes *sync_sb_inodes()* to flush the corresponding dirty pages, then invokes *sync_blockdev()* to explicitly flush the dirty buffer pages owned by the block device that includes the superblock. The *sync_blockdev()* function makes sure that the updates made by *sync_sb_inodes()* are effectively written to disk. The *sync_supers()* function writes the dirty superblocks to disk, if necessary, by using the proper write_super superblock operations. Finally, the *sync_filesystems()* executes the *sync_fssuperblock()* method for all writable file systems.

The *fsync()* system call forces the kernel to write to disk all dirty buffers that belong to the file specified by the fd file descriptor parameter (including the buffer containing its inode, if necessary). The corresponding service routine derives the address of the file object and then invokes the *fsync()* method. Usually, this method ends up invoking the *__writeback_single_inode()* function to write back both the dirty pages associated with the selected inode and the inode itself. The *fdatasync()* system call is very similar to *fsync()*, but writes to disk only the buffers that contain the file's data, not those that contain inode information. Because Linux 2.6 does not have a specific file method for *fdatasync()*, this system call uses the *fsync()* method and is thus identical to *fsync()*.

3.1.2 Modifications to file synchronization mechanism

In our implementation, we intercept *sync()*, *fsync()* and *fdatasync()* system calls and prevent them from flushing data back to the disk. To achieve this functionality, we use a flag based mechanism to switch between standard mode and custom mode of operation. We define *fsync_enabled* flag which is set to false for all explicit synchronization calls. To allow OS to write data back at implicit checkpoints defined previously, we reset the flag to true to allow normal mode of operation. The dual mode of operation also gives the user flexibility to switch between the two data synchronization modes.

Linux usually writes data out of the page cache using a process called pdflush. The behaviour of this multithreaded process is controlled by a series of parameters defined in */proc/sys/vm*. These parameters are accessed and processes in file page-writeback.c. Few important parameters modified in our implementation are described below:

- *dirty_writeback_centisecs* (default 500): In hundredths of a second, this is how often pdflush wakes up to write data to disk.
- *dirty_expire_centiseconds* (default 3000): In hundredths of a second, this parameter defines for how long data can be in the page cache before it's considered expired and must be written at the next opportunity. Note that this default calculates to 30 seconds.
- *dirty_background_ratio* (default 10): Maximum percentage of active pages that can be filled with dirty pages before pdflush begins to write them.

To increase the implicit writeback time, we modify dirty_expire_centiseconds to increase the dirty pages writeback time to 2 minutes under normal load condition. We also modify dirty_writeback_centisecs to 1 minute and dirty_background_ratio to 50 to initiate page flush in between checkpoint duration to handle sudden spikes in workload.

The increase in dirty writeback time from 30 seconds to 2 minutes brings along a considerable risk of data loss in case of system crash between the checkpoints. To mitigate this problem, we modify kernel panic call stack to call *sync()* function before rebooting the system. As we show in the evaluation section, this improves the performance of reads and writes by a factor of 1.4 over conventional Linux kernel.

Overriding application requests to sync data

brings along a considerable risk of data loss in case of system crash between implicit-sync checkpoints. There are different solutions to this problem. One approach is to do a warm reboot, as used in Rio file cache[6], where once the system recovers from a crash, data is read from the file cache and then synced to the disk. Other approach is to write dirty pages sequentially to a separate part of the disk before rebooting and maintain data structures to track location of these pages in disk. Once the system has rebooted, this dump of dirty pages is read and synced to the disk. But, for our usecase, we consider a simplified approach, which is to modify kernel panic call stack to call *sync()* function before rebooting the system.

3.2 Memory Protection

When file data is forced to remain in volatile memory and calls to *fsync()* are overridden, there is an increased responsibility on the operating system to safeguard this data. The major source of concern would be unexpected power failures which would lead to loss of all data stored in the volatile memory. But, in battery backed devices like mobile phones, since there are no unexpected power failures, this is not a source of concern for our system. We flush the dirty pages in the file cache to disk at regular intervals (currently 2 minutes) to reduce the data loss in case the user decides to pull the battery out. To avoid data losses in case the battery runs out of charge, we revert to default behaviour of *fsync()* and flushing dirty pages to disk every 30 seconds, once the battery charge is below a certain threshold.

Though data loss due to power failure is not a major source of concern, the corruption of data kept in memory is a serious issue that needs to be dealt with. As kernel bugs like buffer overflows are not very uncommon [13, 14, 15], the decision to keep data in memory without taking regular backups would make it more prone to corrupt pointer accesses from the kernel. Therefore, a system which risks its data by keeping it in the file cache should also provide a mechanism to prevent the pages from getting corrupted through illegal accesses. A brief description of file permissions, the Linux page cache and how file-backed pages are handled in Linux follows, before describing our mechanisms to provide extra protection to the contents of the file cache.

When user invokes the system call *sys_open()* to open a file, the kernel checks the file permissions and confirms whether the user is authorized to open the file in the mode requested by the user. The file system keeps track of the permissions with which each file was opened, and any subsequent unauthorized calls to *sys_read()* or *sys_write()* are blocked by the file system after checking with these permissions. As this mechanism is implemented at the file system level, it protects the file data only from illegal accesses through the file system calls. However, any corrupt pointer in the kernel which points to the file data page can still corrupt the file data. Therefore a mechanism has to devised which could deal with protection at the page level.

Linux splits the address space of any process into two regions - the user space and the kernel space. While the user space is different for each process and changes with context switches, the kernel address space is shared among all the processes. When a request for loading a page to memory is handled by the system calls, *sys_read()* or *sys_write()*, the page is brought to main memory and mapped to a virtual address in the kernel address space. Needless to say, the page is accessible only while running in Kernel Mode and not from the User Mode. Therefore, the extra protection that we implement deals with accesses from the kernel space and not from the user space.

We propose two different mechanisms through which an increased protection from corrupt kernel pointers can be achieved. The first method write-protects all file-backed pages in the file cache by changing the protection bits in the page table entry, and removes the write protection only when data is to be copied to the file page from the application buffer. The second provides protection by maintaining the file-backed

pages in transcendent memory, which is not directly addressable by kernel.

3.2.1 Write Protection in Page Table entries

A common source of kernel bugs is buffer overflows which would result in illegal pointer accesses to pages from within the kernel. In order to prevent such accesses and allow access to file-backed pages only via file system calls like *sys_write()* and *sys_read()*, a mechanism was needed which would check access rights on each and every access to the page. In most modern systems that support virtual memory, all accesses to the file pages by the kernel are through virtual addresses. Linux has a three-level page table with top level Page Global Directory (PGD), the second level Page Middle Directory (PMD) and the final level containing the actual Page Table Entries (PTE) which stores the physical address of the page frame being referenced. Apart from the physical address of the page, the PTE also stores protection flags that have been set on the page. The protection flags, unlike the ones maintained by the file system, are checked by the virtual-physical address translation hardware, on each access to the page. Therefore, write protecting the pages at the PTE level would protect the page from any illegal accesses to the page. In case of legal writes (i.e. *sys_write()* function calls), the write protection is removed just before copying data from the application buffer into the file page, and restored after copying. This would reduce the window of vulnerability by a great extent.

The following is the code used to write-protect a particular page. The function *page_address()* returns the address of the page we are trying to write-protect. This address (with appropriate masks) is used as index into the PGD, PMD and PTE tables.

```
unsigned long addr;
pgd_t* pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *ptep, pte;

addr = (unsigned long)page_address(page);

pgd = pgd_offset_k(addr);
if (!pgd_none(*pgd))
{
    pmd = pmd_offset(pud, addr);
    if (!pmd_none(*pmd))
    {
        ptep = pte_offset_map(pmd, addr);
        pte = pte_wrprotect(*ptep);
        set_pte_at(&init_mm, addr, ptep, pte);
        flush_tlb_kernel_page(addr);
    }
}
```

The function *pte_wrprotect()* is used to set the write-protect bit of the protection flags maintained in the page table entry. Similarly, for removing write protection before copying data from the application buffer to the *pte_mkwrite()* function is used to remove the write-protect bit from the page table entry. The *set_pte_at()* function writes back the page table entry to the location pointed to by ptep. Finally, the virtual to physical address translation that is cached in the processor TLB is flushed, so that the next access to the page causes the translation mechanism to access the page table so that changes in permission are reflected from the very next access to the page.

It is important to note that the extra protection implemented here is applicable only for pages that are backed by files, and not for any of the anonymous pages that the kernel would have allocated. This is because, unlike the system calls to file backed pages, there is no channelized way in which anonymous pages are accessed.

3.2.2 Using Transcendent Memory

Our second approach for protection of application file data from kernel bugs is to put file-backed pages in a protected area which is not addressable by the kernel. We found that we could leverage transcendent memory [9,10] (as

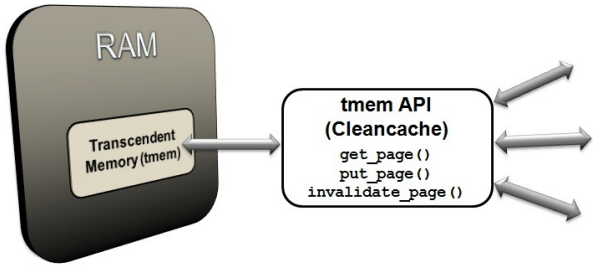

Figure 1: Transcendent Memory

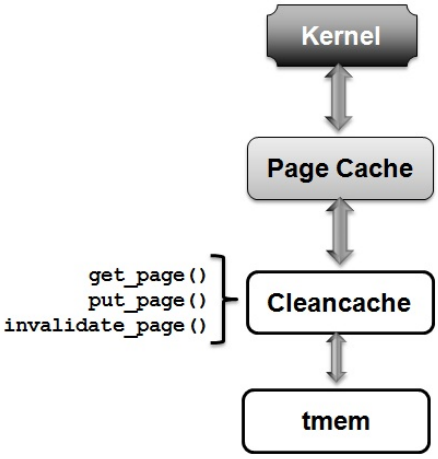

Figure 2: Transcendent Memory: Default configuration

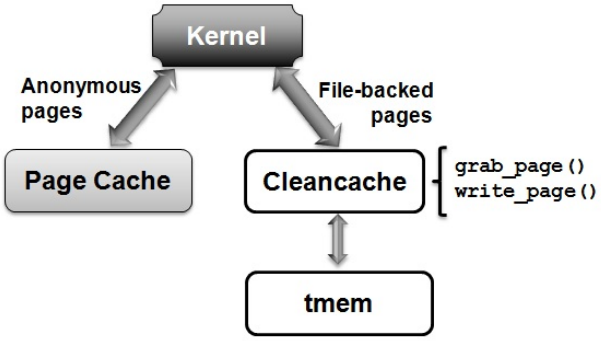

Figure 3: Transcendent Memory: Proposed configuration

shown in Figure 1) for our purpose, as it is a part of memory which is not directly addressable by kernel and can be accessed only through a narrow, well-defined, but a quirky interface called 'cleancache'.

Linux 3.0 kernel started supporting cleancache (as an optional feature), which provides an interface to transcendent memory, with simple calls like *cleancache_get_page()*, *cleancache_put_page()* and *cleancache_invalidate_page()*. Currently, the operating systems view cleancache as a second level cache (Figure 2). So, when the page replacement algorithm evicts a clean page from page cache, it attempts to put the evicted page into transcendent memory using *cleancache_put_page()*. Later, when file system wishes to access a page in a file on disk, it uses *cleancache_get_page()* to check if the page already exists in cleancache, and if it does, the page is copied into the kernel memory, avoiding a disk access. Though this improves the file system performance, the extra memory required for redundant storage could be of concern in systems with limited RAM like mobile phones. In such systems, the file system can use zcache drivers [11, 12] which stores pages in the cleancache in a compressed format.

We intend to use cleancache as a protected area, where we can store the application data to protect it from any corrupt kernel pointers or malicious driver from overriding. Hence, we propose a slightly modified version of cleancache (Figure 3), in which we consider it as the primary cache for file-backed pages. To read or write pages of a file in the cleancache, we extend its interface by adding two APIs - *cleancache_grab_page()* and *cleancache_write_page()*. *cleancache_grab_page()* tries to get a particular page from the cleancache using *cleancache_get_page()* and if the page is not present in the cleancache, the page is brought into cleancache from the disk. *cleancache_write_page()* is used to write to a page in the cleancache at any particular offset within the page: it internally uses *cleancache_grab_page()* to get the page in the cleancache, modifies the page and puts the page back in the cleancache. Android file system code is routed to these interfaces in case of file-backed pages. *cleancache_grab_page()* is called whenever an application wants to read a page of the file, and *cleancache_write_page()* is called when application wants to write to a page of the

file.

In addition to that, we are deviating from the normal use case of cleancache, in which it is used by the system to cache clean (i.e. read-only) pages, which may be evicted at any time later depending upon the working set size of the process and the total physical memory of the system. In our case, since we are using cleancache to store dirty pages as well, the pages are written to disk when they are evicted from the cleancache. Since all file-backed pages are now mapped to an area not directly accessible by kernel, corrupt pointer accesses from the kernel would no longer be able to access the file data.

Another optimization that can be added to the modified cleancache, is the use of zcache drivers to store the pages in a compressed form. This would reduce the overall space used by the file cache and could prove very useful in systems with limited RAM. But there is an extra overhead of compressing and decompressing pages on every read and write to the page.

4 Evaluation

For testing our changes to the file system, we needed a I/O benchmarking tool. We started off by standard benchmarking tools available for Android, namely - AnTuTu Benchmark [20], Quadrant Standard Edition [21], SmartBench [22] and CF-Bench [23]. Most of these benchmarking tools calculate a raw score at the end of tests to show the device performance. However, in most cases, the exact interpretation of these scores was not known and the scores were primarily used as a metric to compare the device with other devices. Moreover, since the underlying implementation of I/O tests was not exposed, we found varying results across different benchmarking tools, some qualifying our implementation as superior while others did not.

To come up with a better implementation of I/O benchmarking tool, we decided to implement our own benchmarking tool. While designing the tool, we came across multiple factors which could possibly affect the I/O performance, which also explains why different benchmarking tools performed differently on the same device. One such factor is the storage structures used by the benchmarking tool.

Android provides several options to save persistent application data. The chosen solution depends on specific needs, such as whether the data should be private to the application or accessible to other applications (and the user) and how much space the data requires[17]. Major data storage options are the following:

- *Shared Preferences*: Store private primitive data in key-value pairs
- *Internal Storage*: Store private data on the device memory
- *External Storage*: Store public data on the shared external storage
- *SQLite Databases*: Store structured data in a private database
- *Network Connection*: Store data on the web

Since external storage is the most common form of storage found in handheld devices, we decided to use external storage for our benchmarking tool. The algorithm for measuring performance is as below:

```
get_start_time()
    repeat for n times
        write to file buffer
        flush the buffer
        issue sync
    end repeat
get_end_time()
```

We ran our benchmarking tests on two separate environments - Android emulator and Samsung Galaxy S3 running Android 4.0.4. Since we did not have access to a Android device with root privileges, we changed the benchmark tool to skip synchronization requests to emulate the behaviour we expect in our synchronization model. Note that doing so will underestimate the time taken to run the benchmark, since there is a fixed amount of cost associated with calling *fsync()* function. But this cost would be

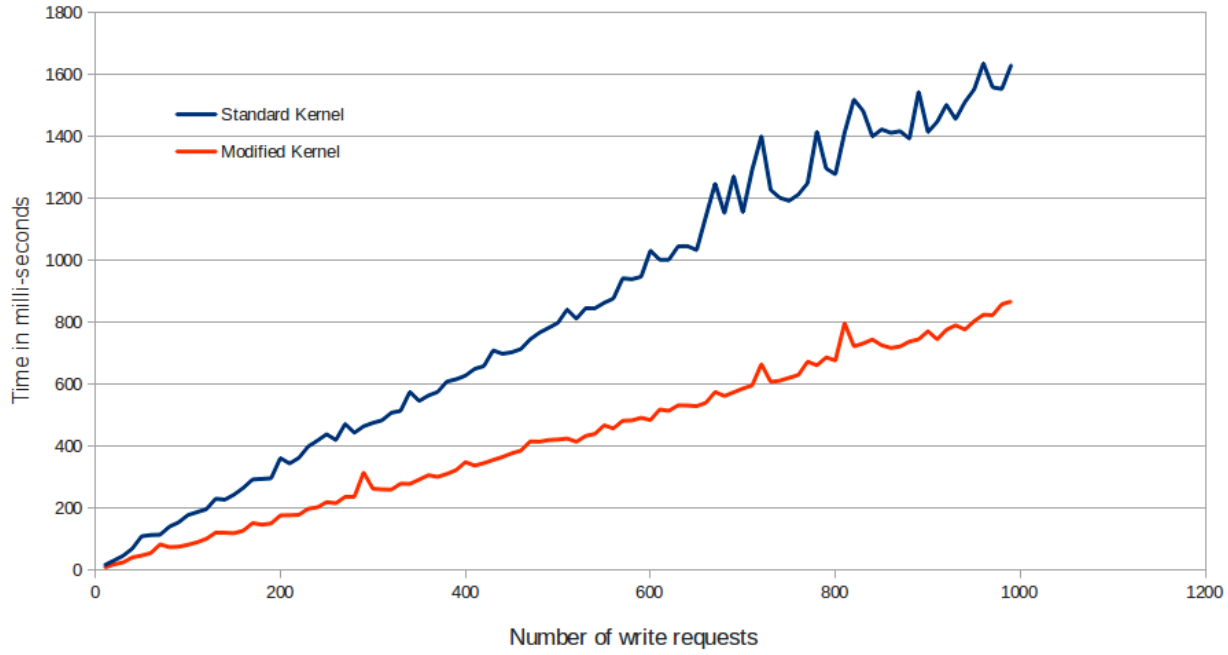

Figure 4: I/O performance on emulated environment

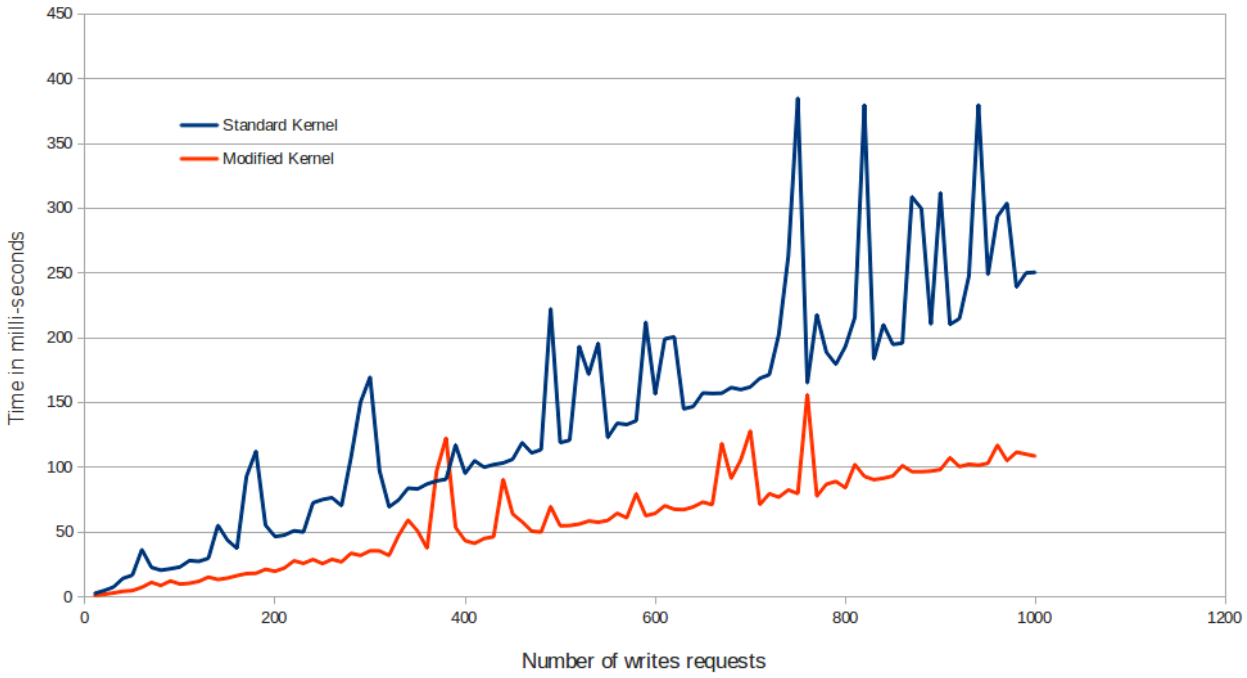

Figure 5: I/O performance on hardware

much lesser than the cost of syncing data to disk. Since our intention is to use real hardware as a proof of concept and not for actual cost/benefit analysis, we do not consider this as a major issue. For more accurate measurements, we planned to run benchmarking tests on Raspberry Pi board. But since a stable port for Android was not available at the time this work

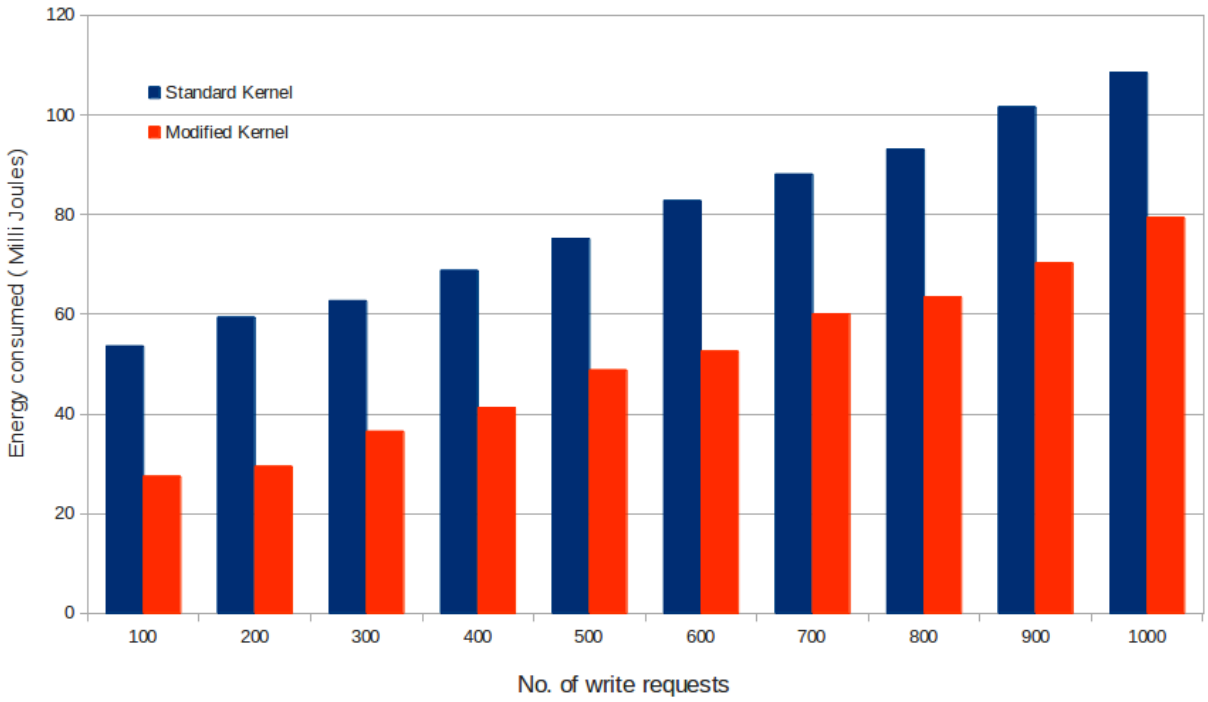

Figure 6: Energy consumption on emulated environment

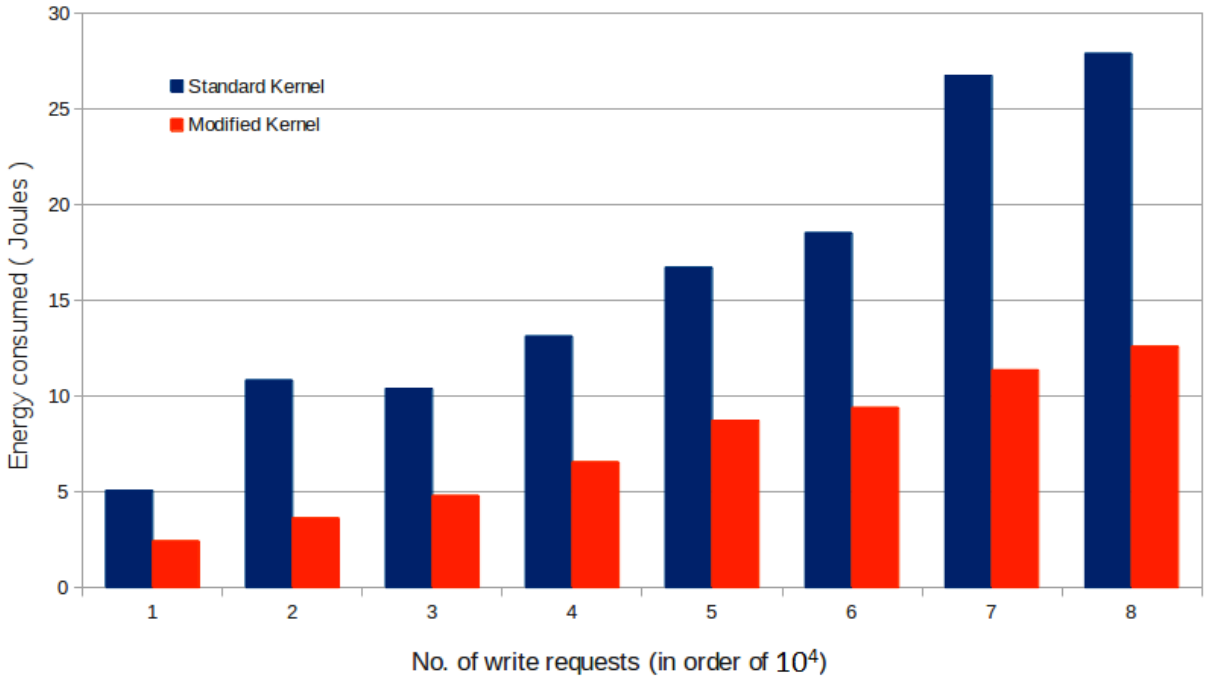

Figure 7: Energy Consumption on hardware

was done, we are unable to present evaluation results for Raspberry Pi hardware.

We present I/O performance of our benchmark on emulated environment and real hardware in

Figure 4 and 5 respectively. Clearly, write intensive workloads perform better in our implementation. We see a performance speedup of upto 143% on emulated environment and upto 155% in real hardware.

An indirect benefit of limiting the disk access is reduction in costs associated with read and write calls[18]. Since getting a page from disk is computationally more expensive operation, it requires more power to fetch a page from disk when compared to getting a page from memory. We test this idea using a power profiling tool called PowerTutor[19]. This tool shows the energy consumed by an application in Joules. Figures 6 and 7 show the result of power profiling using different combination of workloads. As expected, our system outperforms the standard Android kernel and provides significantly cheaper read/write system calls.

An ideal way to evaluate the proposed memory protection mechanisms would be through fault injection. Code to inject various types of faults into the kernel could be written, and our protection mechanism could be evaluated on the basis of how many faults the modified kernel is able to prevent. But this process needed an elaborate study on the various types of faults and ways to inject faults into the kernel. Due to time limitations, we adopt a much simpler method to test our protection mechanism on a per-file basis. During a *sys_write()* system call to write data to the file under test, since address of the page is known and the process is running in kernel mode we could run our test code to access the write-protected page.

5 Conclusions

In our work, we have made a case for persistent memory in mobile operating systems. Our performance experiments show that using techniques defined in the paper, we can achieve considerable performance benefits at little or no risk of data loss. For write intensive workloads, our implementation provides significant performance benefit over conventional Android kernel. By reducing the disk access frequency, we also reduce the cost associated with each read/write request, thereby reducing the power consumption of all applications.

We also define two approaches to protect in-memory data - write protecting pages and using transcendent memory, which provide additional protection to data while it resides in main memory. We believe that combination of persistent memory with protection domains successfully achieves our goal of designing a file cache which is fast and reliable at the same time. Moreover, the ideas presented in this paper are generic and can be applied to any file system implementation.

References

[1] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. ASPLOS 11: Proceeding of the 16th international Conference on Architectural support for Programming Languages and Operating Systems, New York, NY, USA, 2011. ACM.

[2] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. (1992). Non-Volatile Memory for Fast, Reliable File Systems. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), pages 1022.

[3] Jorge Guerra, Leonardo Marmol, Daniel Galano, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. FIU SCIS Technical Report TR-2010-12-01, 2010.

[4] Jason Gait. Phoenix: A Safe In-Memory File System. Communications of the ACM, 33(1):81-86, January 19, 1990.

[5] A.-I.A. Wang et al., Conquest: Better Performance through a Disk/Persistent-RAM Hybrid File System, Proc. 2002 USENIX Ann. Technical Conf., June 2002.

[6] P. M. Chen, W. T. Ng, S. Chandra, et al. The Rio File Cache: Surviving Operating System Crashes. In Proc. of the 7th Int'l. Conf. on ASPLOS, pages 74-83, 1996.

[7] Ng, N. T. and Chen, P. M. 2001. The design and verication of the Rio le cache. IEEE Trans. Comput. 50, 4, 322337.

[8] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. Proc. of the Summer USENIX Conf., June 1992.

[9] D. Magenheimer, C. Mason, D. Mccracken, K. Hackel, and O. Corporation, Transcendent memory : Re-inventing physical memory management in a virtualized environment, OSDI 08 WIP session, 2008, pp. 167-193.

[10] D. Magenheimer and O. Corp, Transcendent Memory on Xen, Xen Summit, 2009, pp. 1-3.

[11] Nitin Gupta. compcache: Compressed caching for linux. http://code.google.com/p/compcache/, Accessed Oct. 2012.

[12] Nitin Gupta, zcache: page cache compression support. http://lwn.net/Articles/396467/, Accessed Oct. 2012.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In 7th USENIX Security Symposium, San Antonio, Texas, January 1998.

[14] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. DARPA Information Survivability Conference and Exposition. January 2000.

[15] David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed System Security Symposium. February 2000.

[16] Android Open Source, Google, 2010. Web. http://source.android.com/source/index.html

[17] H. Kim, N. Agrawal, and C. Ungureanu, Examining storage performance on mobile devices, in Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld), 2011.

[18] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In USENIX, 2010.

[19] Z. Qian Z. Wang R. P. Dick Z. Mao L. Zhang, B. Tiwana and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In Proc. Int. Conf. Hardware/Software Codesign and System Synthesis, 2010.

[20] AnTuTu Labs. AnTuTu Benchmark. http://www.antutulabs.com/antutu-benchmark, Accessed Oct. 2012.

[21] Aurora Softworks. Quadrant Standard Edition. http://www.aurorasoftworks.com, Accessed Oct. 2012.

[22] SmartBench. http://smartphonebenchmarks.com, Accessed Oct. 2012.

[23] CF-bench. CPU and memory benchmark tool. http://www.chainfire.eu/projects/46/CF-Bench/, Accessed Oct. 2012.