SmartCon: Smart Context Switching for Fast Storage IO Devices

JONGMIN GIM, Texas A&M University
TAEHO HWANG, Hanyang University
YOUJIP WON, Hanyang University
KRISHNA KANT, Temple University

Handling of storage IO in modern Operating Systems assumes that such devices are slow and CPU cycles are valuable. Consequently, to effectively exploiting the underlying hardware resources, e.g. CPU cycles, storage bandwidth and etc., whenever an IO request is issued to such device, the requesting thread is switched out in favor of another thread that may be ready to execute. Recent advances in non-volatile storage technologies and multicore CPUs make both of these assumptions increasingly questionable, and an unconditional context switch is no longer desirable. In this paper, we propose a novel mechanism called SmartCon that intelligently decides whether to service a given IO request in interrupt driven manner or busy-wait based manner based on not only the device characteristics but also dynamic parameters such as IO latency, CPU utilization, and IO size. We develop an analytic performance model to project the performance of SmartCon for forthcoming devices. We implement SmartCon mechanism on Linux 2.6 and perform detailed evaluation using three different IO devices: Ramdisk, low-end SSD, and highend SSD. We find that SmartCon yields upto 39% performance gain over the mainstream block device approach for Ramdisk, and upto 45% gain for PCIe based SSD and SATA based SSD's. We examine the detailed behavior of TLB, L1, L2 cache and show that SmartCon achieves significant improvement in all cache miss behaviors.

Categories and Subject Descriptors: D.4.2 [**Operating System**]: Storage Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Non-volatile memory, Solid State Disk, Context switch, I/O Subsystem

ACM Reference Format:
Jongmin Gim, Taeho Hwang, Youjip Won, and Krishna Kant, 2013. SmartCon: Smart Context Switching for Fast Storage IO Devices. *ACM Trans. Storage* 9, 4, Article 39 (March 2013), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The evolution of computer technology has traditionally seen rapid increase in CPU and memory speeds but relatively modest increase in the speed of magnetic disks, which have been the mainstay of secondary storage for quite some time. However, a variety of factors are converging together to change the landscape dramatically. The single core CPU performance began to stall around 2005 [HenkPoley 2014], and the much of the

This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device), and partially supported by IT R&D program MKE/KEIT. [No.10035202, Large Scale hyper-MLC SSD Technology Development]. This research was also supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014- H0301-14-1017) supervised by the NIPA (National IT Industry Promotion Agency).
Author's addresses: J. Gim, Electrical and Computer Engineering, Texas A&M University; T. Hwang and Y. Won (Corresponding Author), Division of Electrical and Computer Engineering, Hanyang University; K. Kant, Dept. of Computer & Info. Science, Temple University.

DOI:http://dx.doi.org/10.1145/0000000.0000000

increase in computing power since then has been in terms of number of cores. This trend is expected to continue in the foreseeable future. This coupled with our inability to effectively parallelize the applications beyond a few cores in most cases [Kasikci et al. 2013] means that we are likely to see many poorly utilized cores on a die. In other words, CPU cycles are no longer quite as precious as they used to be [Ahmadi and Maleki 2010].

On the storage side, the DRAM latency has also stalled, even though the maximum DRAM throughput continues to go up (the so called “memory wall” phenomenon). However, the solid-state non-volatile storage technology has made great strides. Not only do we have relatively mature NAND and NOR flash storage technologies, there is also a host of new non-volatile memory (NVRAM) technologies, with increasing performance, some of which could rival or beat DRAM performance [Burr et al. 2008]. These technologies include SpinRAM, Magnetostrictive RAM (MRAM), Spin Torque Transfer MRAM (STT-MRAM), Ferroelectric RAM (Fe-RAM or FRAM), among others. Fig. 1 shows the read and write latencies for some of the technologies based on the data presented in references [Qureshi et al. 2009; Li et al. 2008; Hosomi et al. 2005; Electronics 2005; Jung et al. 2010]. This convergence between main memory and secondary storage has been well recognized by now [Coburn et al. 2011; Volos et al. 2011; Akel et al. 2011; Venkataraman et al. 2011; Chen et al. 2014; Yang et al. 2012; Lantz et al. 2014] and calls for rethinking of the traditional storage access methods and hierarchies, as discussed later in related work. In this paper, we address an orthogonal issue, that also needs to be reexamined as the storage device speeds increase – namely OS context switches.

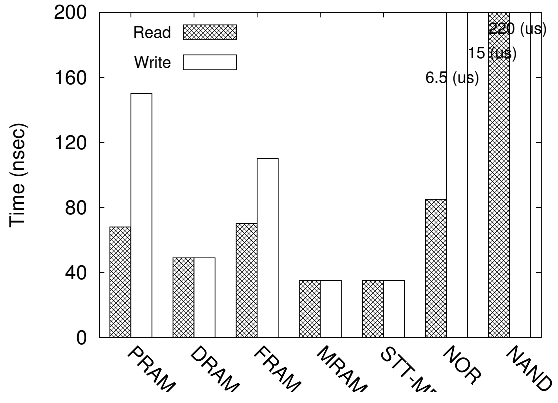


Fig. 1. Access Latency of Memory Devices

Mainstream Operating Systems use context switches to handle IO in order to share valuable CPU cycles among different processes. The convergence of memory and storage, coupled with the emergence of many core CPU demands a more flexible access scheme. In this paper, we discuss an intelligent context switch mechanism called SmartCon to decide whether to switch the context over to another thread in case of storage access or simply stall as done currently for main memory accesses. The switching decision is based upon the static (i.e., dependent on device type) as well as the dynamic (i.e., one that accounts for current conditions such as CPU utilization, IO latencies, etc) attributes of the system.

Although the idea of adaptive context switch is conceptually straightforward, its manifestation in a real system is not. In particular, a straightforward adaptive scheme could result in worse or sometimes significantly worse performance than the current practice of context switch based IO.

The contribution of our work is three fold. First, we develop elaborate context switch decision mechanism which dynamically determines whether to do context switch or not by incorporating the device latency, CPU utilization and IO size. We implement our mechanism in commodity OS (Linux 2.6.) with minimal CPU overhead. Second, we perform comprehensive performance experiments and explore the effectiveness of SmartCon in various respects. We use three representative storage devices: (i) RAMDisk, i.e., secondary storage simulated in a part of DRAM, (ii) a low-end SSD connected to the host via SATA, and (iii) a high-end SSD which is connected to host via PCIe link and has 12 GByte of device cache. To make the experimental results more reliable and comprehensive, we run performance benchmark with both CPU intensive and IO intensive workloads. Third, we develop an analytical performance model that enables us to predict the performance of SmartCon for other non-volatile random access memory (NVRAM) technologies, e.g. PRAM, STT-MRAM, etc.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 examines the various types of context switch overheads and the notion of fast IO device. Section 4 describes the organization of SmartCon. In section 5, we explain the details of SmartCon decision mechanism. Section 6 presents an analytic model for projecting system performance of SmartCon and Section 7 discusses detailed results of our experimental evaluation. Finally, section 8 concludes the paper and points out areas for further work.

2. RELATED WORK

In recent years there has been a tremendous amount of interest in exploring emerging NVRAM technologies, using them in creative ways, and coping with their idiosyncrasies (e.g., very long write times, limited life-time, different write latencies for 0 and 1, etc.). NVRAM technologies that are slower than DRAM but cheaper can be used with main memory semantics with DRAM acting as a cache. Such an arrangement can help reduce memory power consumption, which constitutes an increasing percentage of total platform power [Kant 2008]. Recently, a number of works have proposed new system architecture to exploit PRAM as DRAM alternative or use it as cache [Wu et al. 2009; Lee et al. 2009; Qureshi et al. 2010]. A number of works have examined new types of file systems to exploit the byte addressability and non-volatility of byte addressable NVRAM [Ipek et al. 2009; Jung et al. 2009]. NV-Heap [Coburn et al. 2011] and Mnemosyne [Volos et al. 2011] proposed new types of heap management framework to provide the persistency on the data object in NVRAM while managing NVRAM area as heap. Venkataraman et. al. [Venkataraman et al. 2011] propose new data structure called Consistent Durable Data Structure (CDDS) which uses versioning to provide atomic operation on the byte addressable non-volatile storage. PCRAM based SSD ONYX has shown that it can achieve better performance than high end SSD for IO less than 2 KByte [Akel et al. 2011] IO size.

The above works deal with abstraction issue on byte addressable NVRAM, but do not address the issue of scheduling the accesses on the respective device. The scheduling issue – in particular whether or not to switch context upon IO initiation – is orthogonal and becomes relevant when the NVRAM IO latency is within 1-2 orders of magnitude of main memory access latency. For example, if NVRAM is used as a cache between the high speed DRAM and slow disk levels, a context switch on access to NVRAM may or may not be desirable depending on the NVRAM speed and access size. For concreteness, in the following we evaluate SmartCon in the traditional block storage IO context, but it is clear that the same techniques can be applied to other models of NVRAM access. Jeong et. al. applies busy-wait based IO in Smartphone and shows that

IO throughput increases as much as 13% especially when a number of CPU intensive processes compete for CPU [Jeong et al. 2013].

The context switch behavior has been a subject of extensive research efforts. Ousterhout et. al. [Ousterhout 1990] examine the reason why overall system performance does not scale well with the increase in the hardware speed and conclude that context switch overhead is the main cause for that. Chuanpeng et. al. [Li et al. 2007] show that cache pollution caused by context switch plays a significant role in overall system performance degradation. Kobayashi [Kobayashi 1986] use three typical workloads to analyze the effect of cache pollution in a context switch. Starner et. al. [Starner and Asplund] analyze cache pollution in embedded real-time systems and propose a method to identify its origin. Cache exhibits entirely different behavior under multiprocessor [Agarwal et al. 1988] and multicore environments [Yan and Zhang 2008; Tam et al. 2007]. Yan. et. al. [Yan and Zhang 2008] develop a tool to analyze the interference in the L2 cache shared by multiple cores. Francis et. al. [David et al. 2007] partition the context switch overhead into two: direct cost and indirect cost. Direct cost corresponds to CPU scheduling overhead, time to store and load register values to and from memory, and the time for switching memory map (page table). Indirect cost corresponds to the time for warming up the cache. They find that indirect cost of context switch is an order of magnitude larger than the direct cost. Liu et. al. [Liu et al.] develop a model for direct and indirect cost of context switch and also analyze the relationship between cache miss and context switch. McVoy et. al. develop a method to measure the context switch overhead [McVoy and Staelin 1996].

A context switch not only corrupts the L2 cache but also affects the behavior of TLB (Translation Lookaside Buffer). A number of works attempt to improve the TLB miss overhead caused by context switch. Yamada et. al. [Yamada and Kusakabe 2008] propose to incorporate the overhead of updating address map into CPU scheduling. In selecting the process from ready queue, their CPU scheduler favors the threads that use the same address map as the current thread. This is to minimize the overhead involved in updating the address map. This scheme becomes particularly effective when the system is running multiple applications each of which is designed to exploit thread level parallelism (TLP). Wiggins et. al. [Wiggins et al. 2003] propose a software method to maintain process ID for TLB entry. Venkatasubramanian et. al. [Venkatasubramanian et al. 2009] show that TLB management overhead is one of the most significant performance bottleneck in multicore based VMs and recommend separate tagged TLB for individual virtual platforms.

The issue addressed in this paper also arises in the context of locking; i.e., spin-locks vs. blocking for protecting access to critical regions. The crucial issue is whether it is preferable to spin or block when requesting a lock. Since the optimal strategy depends on the duration of time the lock is held and the overheads involved, several adaptive methods are have been proposed [Karlin et al. 1991]. For example, a simple strategy is to spin for a while and then block, but more sophisticated schemes based on the prediction of lock holding time may show better performance when accurate prediction can be made. Ryan Johnson et. al. [Johnson et al. 2009] make an argument similar to our paper in the case of multi-core processors: as the number of cores increases, the optimal point shifts in favor of spinning.

Another related issue that has long been studied in the OS context is busy-wait vs. interrupt based handling of external events. Interrupts cause uncontrolled context switches thereby resulting in effects such as pollution of cache and TLB which can hurt performance. On the other hand, busy-wait wastes resources. Schemes to address this tradeoff include prioritized busy-wait, interrupt coalescing, and dynamic switching between the two [Salah and Qahtan 2009].

Table I. CPU platform (Core i7 has 8MB 16 way L3.)

Platform	Speed	L1 Cache	L2 cache
Intel Core-i5	2.67 GHz	32 KB (8 way)	256 KB (8 way)
Intel Core-2Duo	1.86 GHz	32 KB (4 way)	4 MB (16 way)
Marvell PXA320	806 MHz	32 KB (32 way)	256 KB (4 way)

Asynchronous IO or AIO is a complementary direction for improving IO performance by attempting to reduce (rather than increase) application blocking for IO completion[The Open Group Base Specifications Issues 6 IEEE Std 1003.1 2003; Elmeleegy et al. 2004; Bhattacharya et al. 2004]. AIO allows the applications to issue multiple IO requests with a single system call and to overlap the IO request with other processing. AIO can eliminate extra threads and can reduce the context switch overhead. Although AIO and SmartCon both strive to make IO more efficient, they are intended for diametrically opposite ends in terms of IO latency. SmartCon is intended for the storage devices with very small latency (e.g., emerging NVRAM technology based storage devices) whereas AIO is intended for storage devices with large IO latency. To reduce the number of context switches, SmartCon avoids context switch by tying up the CPU for a single IO request whereas AIO bundles multiple IO requests in a single system call and has designated thread handle to these requests via select/poll/callback. In this paper we focus on how SmartCon performs against traditional non-bundled IO and leave its comparison against other possibilities including AIO for future work.

3. CONTEXT SWITCH OVERHEAD FOR STORAGE IO

3.1. Direct Overhead

Context switch is categorized into two types [Liu et al.]: (a) Direct overhead, that includes the cost of saving, restoring processor registers, pipeline flush and scheduling, and (b) Indirect overhead, that includes performance degradation caused by L2 cache pollution and TLB warm up. We measure the overhead of context switch for three different platforms shown in Table I.

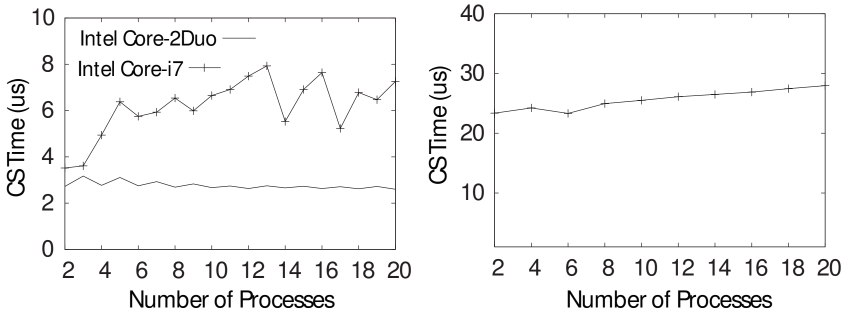
In measuring the context switch overhead, we use `lat_ctx` module of LMBENCH benchmark suite. It creates a number of processes (upto 20), and connects all processes by a ring of Unix pipes. Each process reads a token from its pipe, and writes it to the next -process. Context switch overhead is computed based upon the time interval between writing a token to a pipe by one process and reading the token from the pipe by the next process. This approach is developed by McVoy et.al [McVoy and Staelin 1996] and is widely used to examine the context switch overhead in various platforms.

We compiled `lat_ctx` module with "-O0" option in order to ensure zero sized token. With this definition, the *direct cost* includes all overhead related to switching the process when there is no cache pollution. It includes not only the hardware overhead of saving and restore hardware context, e.g. TLB flush, CPU pipeline flush, saving and restoring register sets, but also includes various software overheads such as switching address maps and crossing the kernel barriers.

Fig. 2 illustrates the results. The direct costs in Fig. 2(a) are at least 30-50 times that of the read cycle of non-volatile memories shown in Fig. 1. For embedded microprocessors as shown in Fig. 2(b), this ratio increases to 250. The context switch overhead becomes relatively more significant for faster storage devices.

3.2. Indirect Overhead: Cache Pollution

Performance degradation caused by context switch mostly comes from the pollution of data cache. To measure this impact, we allocate an integer array of size N Byte, e.g. 512 KByte, to each process. The process computes the sum of all elements in the

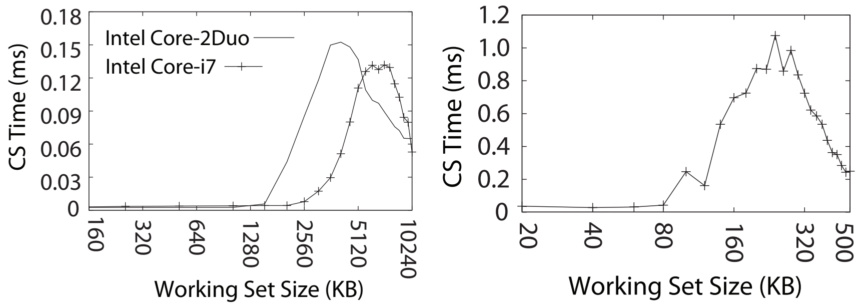


(a) Intel Core-i7 and Core2Duo (b) Marvell PXA 320

Fig. 2. Direct Overhead of Context Switch

integer array and then hands over the CPU to the next process. This operation causes the entire array to be loaded into the cache. We vary the size of this array from 32 KByte to 10 MByte and measure the total time for performing the summation. We use three different CPU's: Intel Core-i7, Intel Core2Duo, and Marvel PXA 320.

Fig. 3 shows the context switch (CS) time as a function of working set size for the 3 CPUs. In Fig. 3, the context switch overhead including direct and indirect components can be as large as 120 us, 150 us, and 1 ms for Intel Core-i7, Intel Core-2Duo and Marvell PXA 320 processors, respectively. The read latency of PRAM, FRAM, MRAM and STT-MRAM corresponds to 68 ns, 70 ns, 35 ns, and 35 ns, which are still smaller than context switch overhead of modern microprocessors. The read latency of NAND flash is 200 μsec [Electronics 2005]. When these devices are used as secondary storage devices, switching the context when performing an IO operation may not justify its overhead. This provides the basis for the smart context switch mechanism explored in this paper.

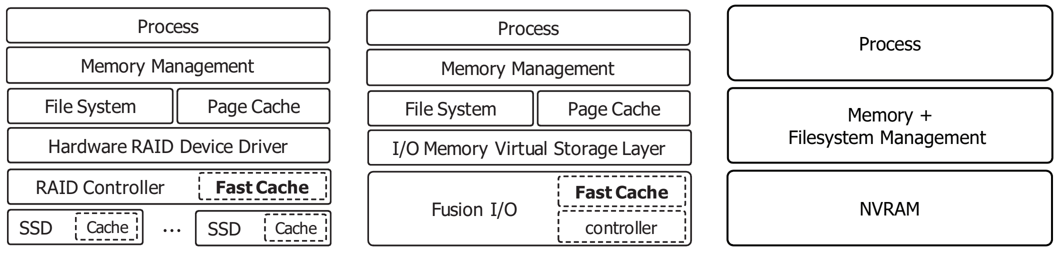


(a) Intel Core-i7 and Core2Due (b) Marvell PXA 320

Fig. 3. Indirect Overhead: Cache Pollution

3.3. Fast Storage Devices

In this paper, we introduce the new term, 'fast' storage device. NAND flash based storage devices, e.g. Solid State Drives (SSD), improve the IO latency by orders of magnitude. High-end SSD's [FusionIO 2010] use PCIe interconnect which is much faster and can be placed closer to CPU. The newly emerging NVRAM technologies such as PRAM, STT-MRAM, and FRAM are expected to be used as storage as well as memory [Freitas and Wilcke 2008]. The access latency of these new types of NVRAM is expected to be comparable to that of DRAM [Samsung Electronics 2007]. Recently, some of the block device controllers, e.g. SSD RAID, are equipped with gigabyte of DRAM as cache and are attached to the northbridge of a platform. The above mentioned devices are orders of magnitude faster and quicker and we call them *Fast IO Devices* as illustrated in Fig. 4. Fig. 4(a) is hardware RAID with tens of gigabyte DRAM cache, is PCIe connected and uses SSD as its storage component [Caulfield et al. 2010]. Fig. 4(b) illustrates high-end NAND flash SSD which is PCIe connected and has several Gbytes of DRAM cache. In Fig. 4(c), Byte-addressable NVRAM is attached to system bus via mainstream DRAM or NOR flash interface and is accessed in the same granularity as DRAM.



(a) Hardware RAID Model (b) Fusion I/O Model (c) Storage Class Memory Model

Fig. 4. Examples of Fast Storage IO Devices

Based upon the type of threads the CPU is handed over to, we can categorize the context switch into three types: (i) between the kernel threads, (ii) between the different user threads, and (iii) self context switch. When a process blocks as a result of IO request, CPU scheduler hands over CPU to another user thread if there is any (type ii) thread. When there is no other process, IOWait daemon (kernel thread) takes over the CPU, stalls CPU until the IO request completes, wakes up the blocked process which has issued the IO request and hands over CPU to the process which is woken up. This is type (iii) and we specifically name it as self context switch.

3.4. Type of Context Switches

There are two types of context switches: *voluntary context switch* and *involuntary context switch* [Liu et al.]. A voluntary context switch occurs when a thread blocks waiting for a resource. Involuntary context switch occurs when a thread has used up its time quantum or higher priority thread has arrived. I/O accesses to most mainstream storage devices, e.g. hard disk, RAID device, SATA based SSD and PCIe based SSD, trigger voluntary context switch. It is worth nothing that ramdisk driver in the mainline Linux kernel [Jones 2006] performs programmed IO and thus voluntary context switch does not occur. To be discussed in detail in the section 7.1, experimental setup, we develop ramdisk driver which handles the IO in context-switch based manner. We modify the existing block device driver [Corbet et al. 2005] so that it can be used with ramdisk and perform context switch [Liu et al.]. We use the modified context switch based ramdisk driver to examine the benefit of busy-wait based IO against interrupt driven IO.

4. ORGANIZATION OF SMARTCON

We carefully argue that as the newer storage device technology, e.g. non-volatile byte addressable memory, and interface technology, e.g. light peak [Intel 2011], PCIe, NVMe [Huffman 2012] emerge, the Operating System should adopt a newer mechanism that determines context switch policy in a more flexible and dynamic manner. We propose a novel IO subsystem mechanism SmartCon, which dynamically determines the IO service policy, context-switch vs busy-wait, properly incorporating the device characteristics as well as the state of the CPU and the IO device for which a request is designated. We design SmartCon with the following three objectives: (i) the decision has to be dynamic; the IO subsystem should apply context switch based IO or busy-wait based IO subject to the type of storage device, e.g. IO latency as well as the state of the system, e.g. CPU utilization, (ii) there should be no side-effect; busy-wait based IO should not affect the performance of the other processes, and (iii) the overhead should be minimal; the overhead of maintaining and probing the state of the system (CPU utilization) and the IO device should be negligible.

In SmartCon, the device type is accessed in every IO operation. The location of the device type critically affects the overall IO performance. We choose to embed device type information in the request queue data structure instead of device data structure. Since request queue is always accessed in every IO request no matter what accessing device type information does not incur additional memory access when it is embedded in request queue. SmartCon IO subsystem consists of three modules: *Device monitor*, *System monitor*, and *SmartCon Decision Module*. Fig. 5 illustrates overall organization of SmartCon IO subsystem.

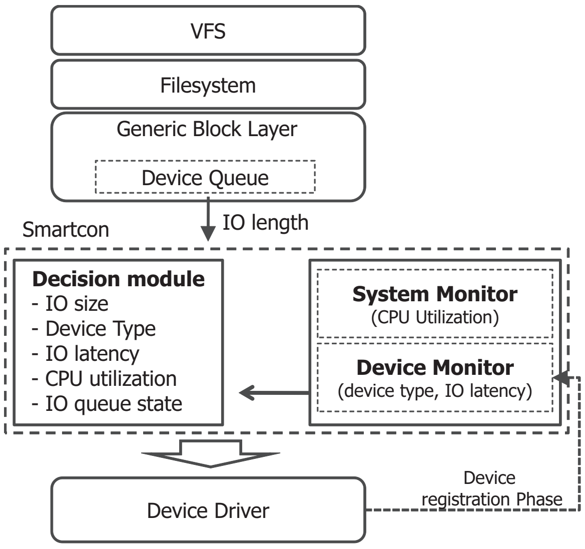


Fig. 5. Organization of SmartCon

4.1. Device Monitor

Device monitor is responsible for determining the IO size threshold and obtaining the device queue length for a given IO. If the IO size is larger than a certain threshold, it is better to switch context to the other process from the system throughput's point of view. For each device, SmartCon establishes threshold value for IO size beyond which IO request is serviced with context switch. When a storage device is added to the system, the Device Monitor examines the IO latencies for different IO sizes from 4 KB to 512 KB with and without context switch, respectively and finds the IO size where the difference between IO latencies with and without context switch respectively, becomes negligible. This probing phase takes less than one minute according to our experiment.

For RevoDrive [OBrien 2013] and Intel X25M G1 [Schmid and Roos 2008], the threshold IO size is 64KB according to our physical experiment.

For a given IO, SmartCon examines the size of a given IO requests as well as the size of all preceding IO requests in the queue. This is because SmartCon needs to estimate the total time a given IO request for which the CPU will be blocked for completing the IO request.

SmartCon categorizes the IO device into three types: *very fast*, *fast* and *mainstream*. IO for mainstream device is always serviced with context-switch. IO for very fast device is always serviced with busy-wait. IO's for fast device are serviced either by busy-wait or by context-switch subject to the decision of SmartCon. Typical example for 'very fast' device can be ramdisk. Since ramdisk IO is implemented via memory copy operation, it is always beneficial to service ramdisk IO in busy-wait mode. The non-volatile RAM devices, e.g. STT-MRAM, FRAM, PC-RAM and etc some of which are already commercially available, can be good candidates for very fast device.

In Linux, each storage device is allocated a request queue. In the current implementation of Linux 2.6, 4 byte data structure is allocated to each queue to represent the status of the queue and only 22bits are being used. We allocate two bits for SmartCon decision from the unused portion of the request queue status data structure. These two is to denote whether a given device is '*mainstream*' and whether a given device is '*very fast*'.

4.2. System Monitor

SmartCon establishes a threshold value for CPU utilization. The *System Monitor* examines the CPU utilization if there are sufficient CPU cycles available for stalling. This is to prohibit the SmartCon enabled IO subsystem from monopolizing the CPU and from degrading the overall system performance.

The CPU threshold is governed by a number of factors, the most important one being the number of available hardware threads. We measure the CPU utilization for busy-wait driven IO in Core-i5. We read 256 MB file from RevoDrive with varying IO sizes from 4Kbyte to 64 KByte. Table II shows the result. Busy-wait driven IO entails 25% CPU utilization in Core-i5 and 44% for Intel Core2Duo, respectively. For RevoDrive device, SmartCon sets threshold for CPU utilization as 75% and 55% for Intel Core-i5 and Intel Core2Duo, respectively.

There are two key implementation issues in system monitor. First, measuring the CPU utilization should not entail any significant overhead. Second, we need to determine the proper length of the measurement window. The current `CPU_Stat` structure of the Linux Kernel carries four counters for CPU usage: IOWait, Idle, System, and User. For each timer interrupt, one of these fields is incremented and the CPU utilization is computed as the ratio of the increment in a given field against the sum of the increments for all fields. Since we examine CPU utilization for every IO request, special care needs to be taken to make the overhead of measuring and computing CPU utilization minimal. We find that the CPU overhead of adding up all four counters in every IO request is prohibitively large, which makes the SmartCon dysfunctional. We introduce on additional 64bit counter `total_CPU_stat` which is incremented for every timer interrupt. CPU utilization is obtained by dividing the respective counter by `total_CPU_stat`. The additional overhead caused by CPU monitoring activity of System monitor corresponds to one additional division operation for each timer interrupt. Most of the kernel data structures are designed with cache line size in mind. The existing `CPU_Stat` structure is 56 bytes long; therefore, with an 8 byte `Total_CPU_stat` field, the CPU statistics object is still cache aligned.

Determining the right observation window size is of critical importance for SmartCon to function properly. With coarse grained observation, SmartCon may not be able to properly capture the IO and CPU load on the system and cannot make right decision on context switch. The CPU utilization of Linux kernel is updated at every timer interrupt. The default resolution frequency of the timer interrupt in Linux kernel is 100 or every 10 msec. We use the default one since 10 msec resolution is sufficient for most applications and increasing the interrupt frequency puts higher load on the system. We run the experiment a number of times with different observation window sizes: from 50 to 1000ms. Observations become noisy and the observation accuracy gradually decreases with smaller window size. According to our experiment, we find that window size of 100 msec is a reasonable compromise between the accuracy, system load and the sampling noise.

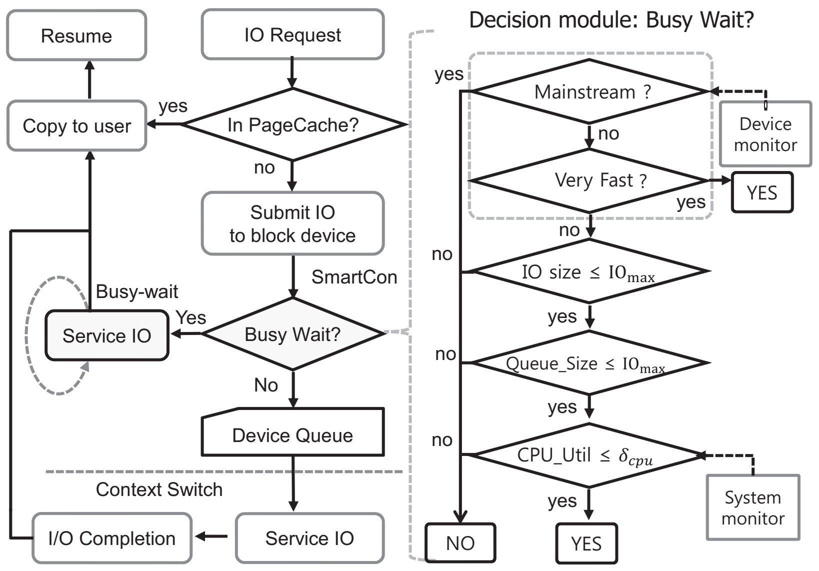


Fig. 6. SmartCon Overview

5. SMARTCON DECISION MECHANISM

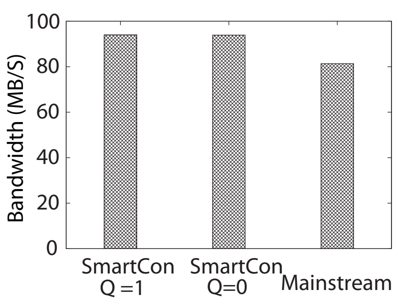
5.1. Algorithm

The *SmartCon Decision Module* collects the system information from Device Monitor and System Monitor. It examines the IO size and decides whether to switch context or not for a given IO operation.

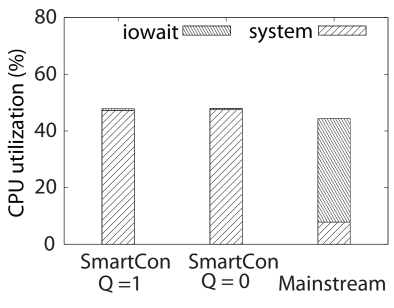
Fig. 6 illustrates the flow of SmartCon enabled IO subsystem. The smartCon decision module examines four conditions. SmartCon carefully lays out the order in which the individual conditions are examined so that the overall overhead of examining the conditions are minimized.

SmartCon first examines two bits to quickly rule out mainstream devices and very fast devices. It then examines the IO size and the queue length of the device, which further eliminates cases where context switch will occur routinely. The CPU utilization is examined in the last step since this is the most time consuming task among the four.

SmartCon incorporates the overhead of servicing backlog of the IO requests in making decision for context switch. SmartCon examines the request queue and computes total IO size for the requests in the queue. If total IO size is greater than IO size threshold, SmartCon performs context switch to service the respective IO request.



(a) Read Performance



(b) CPU Utilization

Fig. 7. Device-load aware Context Switch under different max IO sizes, (Q: Queue Length)

Table II. Threshold for Core-i5 using RevoDrive

IO Size	user (%)	sys (%)	total (%)	Max$_{CPU}$(%)
4KB	0.06	25.03	25	75
8KB	0.07	25.02	25	75
16KB	0.06	25.03	25	75
32KB	0.05	25.06	25	75
64KB	0.02	25.07	25	75

We ran IOzone benchmark with 4KB IO size and examined the number of outstanding requests in the request queue with Intel X25M G1. Fig. 7(a) shows the Read performance and Fig. 7(b) shows the CPU utilization for 3 cases: (a) Smartcon that disables busy wait mode if the device has at least one other IO pending or in progress when the IO is initiated, (b) Smartcon that does busy wait only when there is no other IO in progress when the IO is requested, and (c) Mainstream (i.e., no busy wait). Average request queue length is 1.09. In Fig. 7(a), by incorporating device-load aware context switch, SmartCon achieves 15% performance increase. 'Q' in Fig. 7 denotes the maximum request queue length (as seen by the arriving IO request) beyond which the SmartCon disables busy-wait mode. We find that for Q=0 and 1, SmartCon yields the almost identical performance and CPU utilization.

5.2. Overhead of SmartCon

Busy-wait based IO consumes more CPU cycles than interrupt driven IO does. We examine the CPU overhead caused by busy-wait based IO. We read the file sequentially with context switch based and with busy-wait based IO and compare the CPU utilization. We use PCIe attached high performance SSD (RevoDrive3) and use direct IO option to disable the filesystem prefetch and the buffer cache. Fig. 8 illustrates the results. There are two important messages from Fig. 8. First, busy-wait based IO service indeed consumes significantly more CPU cycles than Context-switch based IO service ranging from twice the CPU cycle for 4KByte to 8 times the CPU cycles for 512KByte, respectively. Context switch based IO becomes more CPU efficient with larger IO size since larger amount of data is transferred per context switch. Second, when there are sufficient amount of available CPU cycles, the available CPU cycles can be used to improve the overall system performance. In context switch based IO service, CPU cycles used by `system` and the `IOwait` thread together denote the total CPU cycles involved in servicing the IO request. From total CPU utilization in context switch based IO service, significant fraction of CPU cycles (from 25% up to 45%) is consumed by `IOWait`.

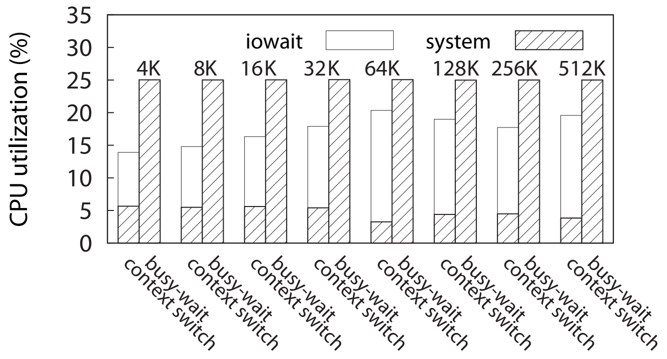


Fig. 8. CPU utilization: Seq. Read in mainstream and SmartCon IO (RevoDrive)

6. ANALYTIC MODELING OF SMARTCON

One of the critical concerns for practitioners in using NVRAM device is to predict the IO performance for different latency NVRAM devices. For some slow NVRAM device, it might be better to use context switch based IO rather than busy-wait based IO.

We develop analytical model which project the performance of a given NVRAM device. The Cycles Per Instruction (CPI) is a key metric for analytic modeling and can be expressed in terms of platform and workload parameters [Kant and Won 1999]. The context switch behavior impacts the cache behavior and hence the CPI. It also adds to memory access latency and may also increase *path length*, i.e., number of instructions executed per user transaction. The CPI can be formulated as in Eq. 1.

$$CPI = CPI_{base} + P_{L2} * \tau_{mem} + \tau_{TLB} \tag{1}$$

where CPI_{base} represents the base CPI (i.e., CPI with infinite amount of L2 cache and no need to go to the memory); P_{L2} denotes the L2 cache miss probability; τ_{mem} and τ_{TLB} are the effective memory access latency and effective TLB access latency, respectively. Effective memory access latency incorporates the average overhead of fetching the respective page from the storage device, when the targeted cacheline is not available in DRAM.

CPI_{base} can be further broken down as in Eq. 2

$$CPI_{base} = CPI_0 + \tau_{cache} \tag{2}$$

where CPI_0 denotes the cycles per instruction and τ_{cache} denotes the effective latency of accessing instruction and data cache at L1 and L2 level. τ_{cache} can be formulated as $\tau_{cache} = \tau_{L1} + P_{L1} * \tau_{L2}$. The effective memory access latency, τ_{mem} in Eq. 1 can be formulated as Eq. 3.

$$\tau_{mem} = \tau_{mem0} + P_{mem} * \tau_{IO} \tag{3}$$

where τ_{mem0}, P_{mem}, and τ_{IO} denote memory access latency for a cacheline, memory miss probability, and the access latency of the IO device, respectively. The unit of memory access is a cacheline (64B in our system), and the unit of storage device access is a page size (4KB in our system). Because we set the experimental environment as sequential workload and 4KB IO size, the P_{mem0} can be simply calculated as 1.6%, (64B/4096B)*100. The IO access latency in Eq. 3, τ_{IO}, can be formulated as in Eq. 4.

$$\tau_{IO} = \tau_{nvram} + P_{ctx} * \tau_{ctx} \tag{4}$$

where τ_{nvram} denotes the base access latency of the NVRAM device, and τ_{ctx} and P_{ctx} denote the context switch overhead and the probability of context switches, respectively. Since the experiment was done in ramdisk, the τ_{nvram} is same to τ_{mem0} *

Table III. Parameters for Analytical Model, (Each τ_{nvram} corresponds to τ_{mem0}*64 for the DDR clock)

ITEM	SmartCon	Mainstream
CPU Utilization	32.9%	35.7%
Normalized Path Length (l)	1	1.14
CPI (CPI_0)	1 cycle	1 cycle
L1 latency (τ_{L1})	3 cycle	3 cycle
L1 miss prob. (P_{L1})	0.115	0.244
L2 latency (τ_{L2})	14 cycle	14 cycle
L2 miss prob. (P_{L1})	0.000255	0.000324
TLB latency (τ_{TLB0})	2.0 cycle	2.0 cycle
TLB miss prob. (P_{TLB})	0.124	0.133
Context Switch overhead (τ_{ctx})	1488 cycle	1488 cycle
Context Switch prob. (P_{ctx})	0.00316	0.00778
Memory miss prob. (P_{mem})	1.56	1.56
DRAM 533MHz (τ_{mem0})	111.88	111.88
DRAM 800MHz (τ_{mem0})	74.4	74.4
DRAM 1067MHz (τ_{mem0})	55.94	55.94

64; where the cacheline size and the block size correspond to 64byte and 4KByte. The effective TLB access latency, τ_{TLB} in Eq. 1 can be formulated as in Eq. 5.

$$\tau_{TLB} = \tau_{TLB0} + P_{TLB} * \tau_{mem0} \tag{5}$$

where τ_{TLB0}, P_{TLB}, and τ_{mem0} denote the TLB access latency, TLB miss probability, and the latency of accessing a memory cache-line, respectively.

Based upon overall CPI, path length (the number of instructions) and the CPU utilization, we can obtain or project performance ratio for different storage systems (under the same workload).

Let U denote the CPU utilization, ℓ the *path length*, or the number of instructions per user level transaction, and f the processor frequency. Then the transaction rate supported by the CPU is given by $U \times f/(CPI \times \ell)$. Consequently, for two variants of the system, say i and j, the relative performance (ρ) is given by:

$$\rho = \frac{CPI_i}{CPI_j}\frac{\ell_i}{\ell_j}\frac{U_j}{U_i} \tag{6}$$

The calibration parameters for this model were obtained by running IOZONE benchmark (sequential read, 256 MB file size, 4 KB IO size) on the Intel Core2Duo (1.86 GHz) platform with 2 GB of DRAM. We ran the experiment five times and take the average. In each run, we rebooted the target computer to minimize variability in the cache hit rate. The resulting parameters are shown in Table III. Although most of these parameters remain unaffected by the NVRAM storage device in use, some would change. In particular, the CPU utilization would change, but we assume that the fractional change remains invariant within the range of NVRAM device parameters that we consider. Similarly, although the actual path length will change, we again assume that the fractional path length change remains as shown – namely, 14% lower for SmartCon. Based on these parameters and assumptions, we attempted to validate our model against the measured results for RAMDISK. For comprehensive validation, we ran physical experiments under three different DRAM clock settings as shown in Table IV.

According to Eq. 6, the model predicts relative performance of 1.23 - 1.24 for SmartCon, whereas the experiments show improvements of 1.21 - 1.23, for 533Mhz, 800 Mhz

Table IV. Measured and Projected Normalized Performance of SmartCon against mainstream Context Switch based IO

RAMDISK Speed (DDR Clock)	Model Performance	Measured Performance
533 Mhz	1.24	1.21
800 Mhz	1.23	1.22
1067 Mhz	1.24	1.23

Table V. Characteristic of Devices used in our experiment

	Ramdisk	SSD	RevoDrive
Model	DDR2 800Mhz	Intel X25M G1	OCZ OCZ revodrive 3 x2
Capacity	2GB	80GB	480GB
Cache Size	NA	16MB	NA
Byte addr	Y	N	N
Data Trans.	CPU	DMA	DMA
Max Bdw	1.2GB/s	250MB/s	1500MB/s
Interface	DIMM	SATA2	PCIe 2.0

and 1067 Mhz DDR clock settings, respectively. This is an excellent match for a very simple model proposed here, and we believe that the model is adequate to estimate performance improvements if the RAMDISK were to be replaced by real NVRAM based storage with characteristics similar to those for DRAM.

7. PERFORMANCE EVALUATION

7.1. Experimental Setup

Smartcon was implemented on Block Device Layer of Linux Kernel 2.6.32 (Ubuntu 10.01). In SmartCon, OS adopts different action after OS inserts the request to the request queue (`make_request`). In context switch based IO service, OS calls `IO_schedule()` and the process is switched as a result. `IO_schedule()` is the kernel function of Linux OS. It saves the registers of the currently running thread and inserts the thread to the set of blocked processes. In busy-wait IO, the OS skips calling `IO_schedule()`. Then, in both cases, the OS dispatches the IO request to the device via calling `_generic_unplug_device`.

We examine the performance of SmartCon IO subsystem for three different storage devices each of which is carefully chosen to represent the characteristics of the different categories of storage devices: RAMDISK [msdn 2010], low-end SSD [Park and Shen 2009], and high-end SSD [FusionIO 2010]. To examine the effect of CPU clock speed and the number of CPU cores, we use Intel Core2Duo for low-end SSD and Core-i5 platforms for high-end SSD, respectively. Table V illustrates the device characteristics.

To ensure comprehensiveness of our performance study, we use three benchmarks, IOZone, Postmark [Katcher 1997], and Filebench [McDougall and Mauro 2005]. IOZone and Postmark benchmarks are used to generate data intensive IO and metadata intensive IO, respectively. Using postmark, we examine the number of file creations per second, varying the size of created files. Filebench is used to generate more realistic file workload: file server workload, multi stream read, single read and mail server workload.

Since SmartCon incorporates CPU utilization and IO queue length, IO size and IO latency in the decision process, it is important to examine the performance under various system loads and IO sizes. We perform each set of experiments under three different system loads.

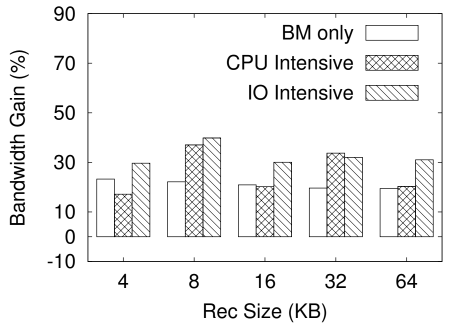
• BM_{only}: we use the system without any ongoing process other than benchmark application. In most case, most of the context switches will be self context switch (type iii).

• $CPU_{Intensive}$: we examine the performance of SmartCon enabled IO subsystem when the CPU is saturated. We run ten number crunching applications to saturate the CPU. This is to examine if the SmartCon properly incorporates the current CPU utilization to make context switch decision.

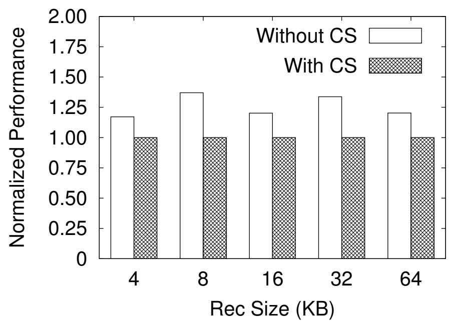
• $IO_{Intensive}$: we examine the performance of SmartCon enabled IO subsystem when the respective storage device is saturated. SmartCon examines IO queue length and applies busy-waits only when the respective IO device is not busy. For this purpose,we run four IO intensive applications alongside the benchmark.

The initialization module of SmartCon probes the device and CPU and determines the threshold value for CPU utilization and IO size. IO size threshold is set to 64 KB. CPU utilization threshold value is set to 85% for RevoDrive SSD on Core-i5 and 53% for Intel SSD on Core2Duo, respectively. We measure IO performance, number of context switches, and CPU utilization for each case. Here, IO performance is defined as the read bandwidth in MB/sec. Therefore, the Bandwidth Gain shown in the figures corresponds to the additional read bandwidth provided by using SmartCon.

7.2. Ramdisk



(a) IOZONE, Sequential Read, 256 MByte file,

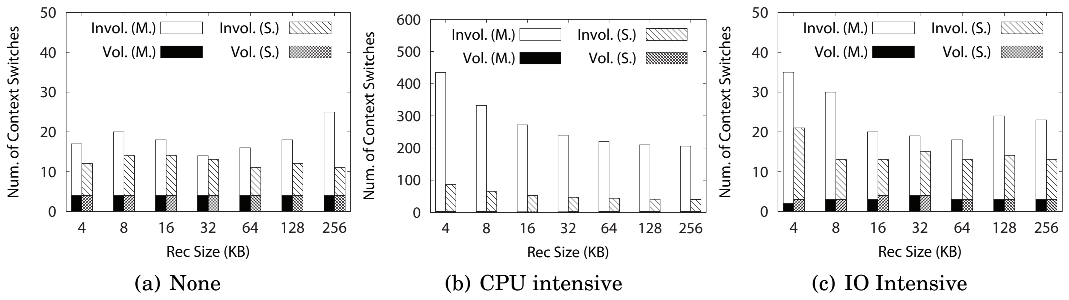


(b) Read Performance With/without Context Switch (NP 1.0 in Y axis denotes I/O perf. of Mainstream)

Fig. 9. Ramdisk

We first examine the performance of RAMDISK device. This is to project the performance gain which SmartCon can bring when the future non-volatile RAM with DRAM-like latency, e.g. STT-MRAM [Li et al. 2008] is deployed as a storage device. We ran IOZONE benchmark (sequential read, 256 MByte file) and examined the sequential read performance under different IO sizes (from 4 KByte to 256 KByte). The experiments were performed under three different systems loads: BM only, CPU intensive and IO intensive. We turn off filesystem prefetching. Fig. 9 illustrates the read performance improvement achieved by SmartCon compared to context switch based IO subsystem.

In Fig. 9(a), when there is no other process, SmartCon yields approximately 20% improvement for all IO sizes. This confirms that significant fraction of IO latency is due

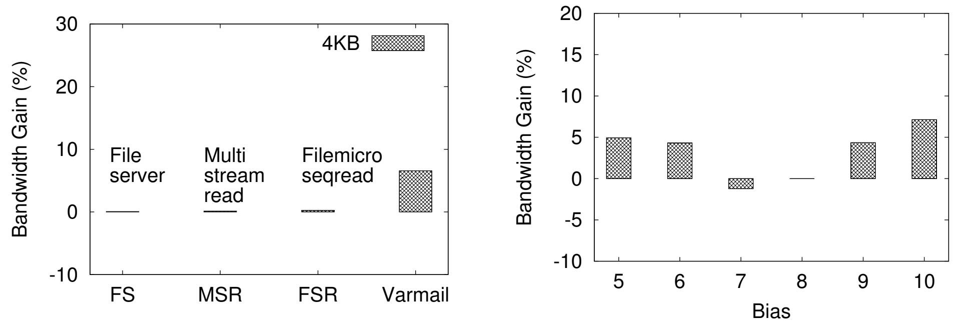


(a) None (b) CPU intensive (c) IO Intensive

Fig. 10. Ramdisk: Number of Context Switches (M.: Mainstream, S.: SmartCon)

to the overhead caused by context switch. The benefit of SmartCon becomes more significant ($>$ 30% performance gain), when IO subsystem is highly loaded with other IO bound processes ($IO_{Intensive}$). This is because when a request from a process is serviced by busy-wait based IOs, the process does not have to compete with other processes to acquire CPU when the IO completes.

We can find the reason for performance advantage of SmartCon by examining the number of context switches. Fig. 10 shows the number of voluntary and involuntary context switches. The number of voluntary context switches remains almost the same in Mainstream IO and SmartCon IO. However, the number of involuntary context switches differs a lot. In CPU intensive environment, the most context switches in mainstream IO subsystem are *involuntary* because of the presence of higher priority processes which the RAMDISK driver will switch in when I/O is complete. In the SmartCon IO subsystem, RAMDISK driver holds CPU till the timer interrupt occurs. The number of involuntary context switches in SmartCon is approximately 1/6th of that for the mainstream IO.

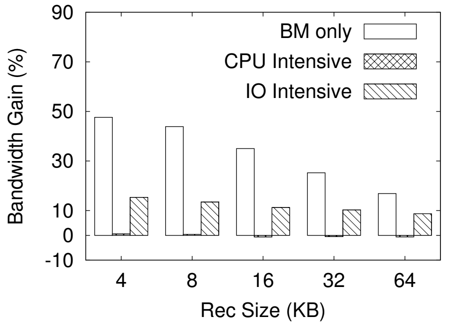


(a) Filebench, 1000 files, 5 IO threads (b) Postmark, file creation, file size = from 4 KB to 64 KB, bias is (number of reads)/(number of writes)

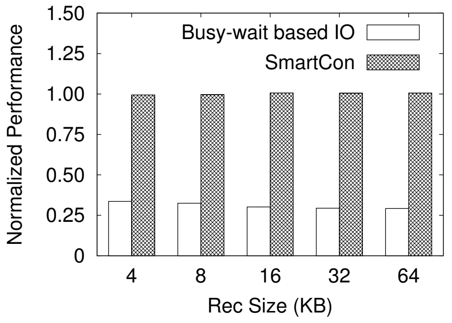
Fig. 11. Ramdisk: Read Performance on Filebench and Postmark

Fig. 11(a) illustrates the performance gain in SmartCon in four workloads. There is practically no performance gain in the first three workloads: File server, Multi stream read, and File Micro read. Mail server workload yields 8% performance gain in SmartCon. File Server, Multistream read and single stream read generate large size sequential IO (1 MByte) and therefore the advantage of busy wait based IO becomes marginal. On the other hand, mail server workload generates 16 KByte random IO which results in significant performance improvement in SmartCon. Fig. 11(b) illustrates performance results of Postmark. Here, the SmartCon advantage varies with bias and is as large as 7 %.

7.3. High-End SSD



(a) IOZone, seq. read, 256MB file



(b) Read performance with and without context switch (NP 1.0 in Y axis denotes I/O perf. of Mainstream)

Fig. 12. OCZ RevoDrive3, 480GByte

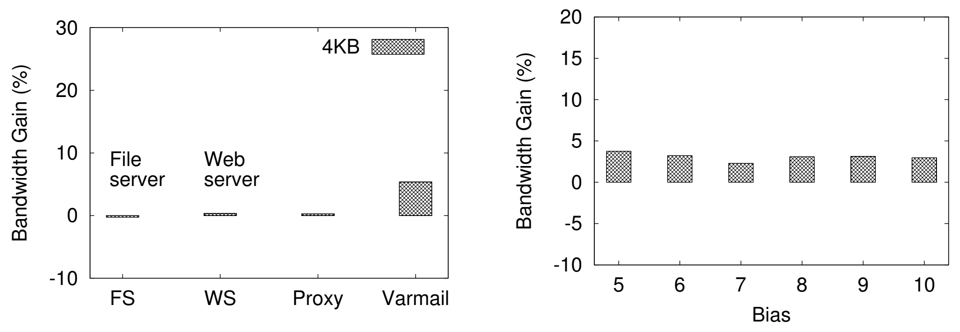
We examine the performance of OCZ RevoDrive SSD with IO size threshold and CPU threshold set to 64 KByte and 75%, respectively. Fig. 12(a) illustrates the performance for IOZONE benchmark. For 4 KByte IO size, SmartCon yields 48%, 1%, and 16% bandwidth improvement for BM_{only}, $CPU_{Intensive}$, and $IO_{Intensive}$, respectively. The improvement becomes less significant for larger IO. Since SmartCon and Mainstream IO subsystem show identical performance when the IO size is greater than IO size threshold (64KB).

In the IO intensive environment, the CPU utilization is found to be 62% - 68% for all record sizes. The SmartCon module decides to stall CPU only when IO size is less than or equal to 64 KB. Fig. 12(a) shows that SmartCon achieves up to 16% improvement in IO bandwidth.

For BM_{only} environment, the CPU utilization stays at 75% and SmartCon stalls CPU. When background process performs CPU intensive task, SmartCon does context switch on every IO operation. Thanks to CPU utilization aware context switch mechanism, SmartCon monopolizes the CPU core only when there are sufficient CPU cycles left. Fig. 12(b) illustrates that SmartCon exhibits robust performance and yields as good a performance as mainstream IO when the CPU is saturated.

Fig. 13 shows the performance result of Filebench and Postmark with RevoDrive. In Fig. 13(a), SmartCon shows 6% bandwidth improvement on Varmail, while it does not bring any improve in the other three workloads due to the same reason as in Fig. 11. In the Postmark, the performance gain of SmartCon is less significant in RevoDrive (less than 4%) than in Ramdisk (5% - 7%) (Fig. 13(b)). We measure the read performance of SmartCon under Bonnie++. Fig. 14 shows the results. SmartCon achieves 45% to 65% read performance improvement for Revodrive and Intel X25M G1.

We examine how SmartCon affects the CPU performance of the other processes. One of the important design objectives of SmartCon is to minimize the side effect of busy-wait driven IO to the other processes. To this end, we ran CPU intensive benchmark Dhrystone [Weicker 1984] with and without SmartCon, respectively and measured the



(a) Filebench: 1000 files, 5 IO threads

(b) Postmark: file creation, file size = 4KB to 64KB, bias is (number of reads)/(number of writes)

Fig. 13. RevoDrive: Read Performance on Filebench and Postmark

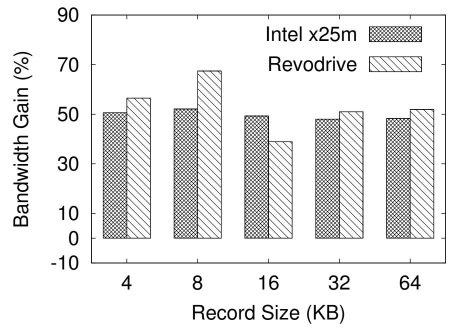


Fig. 14. Bonnie++: Read Performance for RevoDrive and Intel X25M SSD

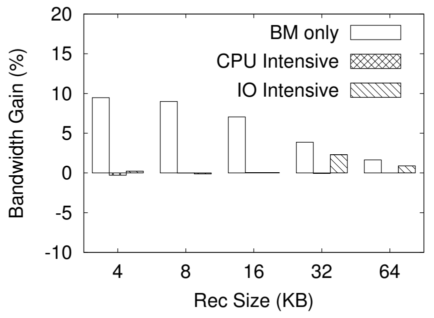
Table VI. Drystone Performance with SmartCon and Mainstream IO

Time	Mainstream	SmartCon	Gain
real	83m 48s	82m 39s	1.4 %
user	36m 24s	37m 44s	-3.5 %
sys	35m 44s	36m 54s	-3.1 %
IOwait	11m 40s	8m 1s	32 %

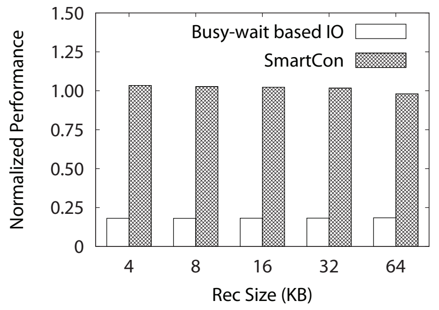
benchmark performance. To generate IO requests, we ran IOZone concurrently. The total execution time of Dhrystone decreases by 1.4% in SmartCon (Table VI). Thus, SmartCon does not negatively affect the performance of the other processes in the system. SmartCon consumes more CPU cycles: `user` and `system` part of CPU utilization increase by 3% in SmartCon. However, it significantly reduces the IO waiting time (32%). As a result, overall CPU throughput remains almost unaffected.

7.4. Low-End SSD

We ran the same set of experiments with low-end SSD (Intel X25M G1) and two core CPU (Intel Core2Duo). In Intel Core2Duo, there are only two CPU cores and the clock rate is slower than for Core-i5. Therefore, we need to reserve a relatively larger fraction of CPU cycles to execute SmartCon. Device Monitor and System monitor set IO size threshold and CPU threshold value as 64 KByte and 55%, respectively.



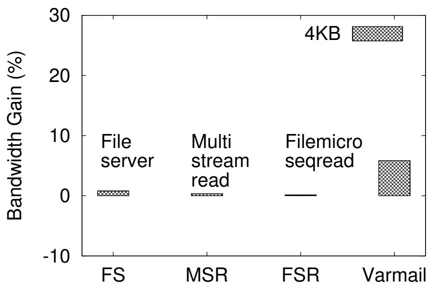
(a) IOZONE, Sequential Read, 256 MByte



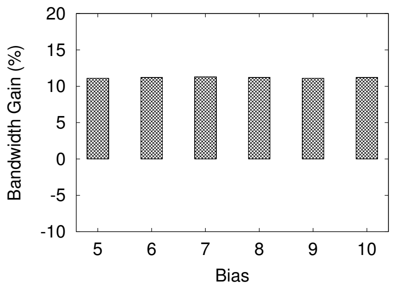
(b) Read Performance with and without context switch (NP 1.0 in Y axis denotes I/O perf. of Mainstream)

Fig. 15. Intel X25M G1

Fig 15(a) illustrates normalized performance of SSD between mainstream and SmartCon IO. Clearly, the advantage of using SmartCon is not as significant in relatively slow storage device as in highend SSD's and Ramdisk. For BM_{only} environment, CPU utilization is lower than CPU threshold and therefore SmartCon stalls CPU. SmartCon yields up to 9% performance gain in IO bandwidth. In $CPU_{Intensive}$ and $IO_{Intensive}$, CPU utilization is higher than CPU threshold value. SmartCon switches context in every IO and yields the identical performance as mainstream IO. We examine the effectiveness of conditional blocking mechanism of SmartCon. Fig 15(b) illustrates the normalized performance of SmartCon against the one where CPU *always* stalls waiting for an IO completion. The performance becomes only 25% of that of SmartCon.



(a) Filebench, 10000 files, 10 IO threads, 4 KB IO



(b) Postmark, file creation, file size = from 4 KB to 64 KB, bias is (number of reads)/(number of writes), (Write Performance: 9%)

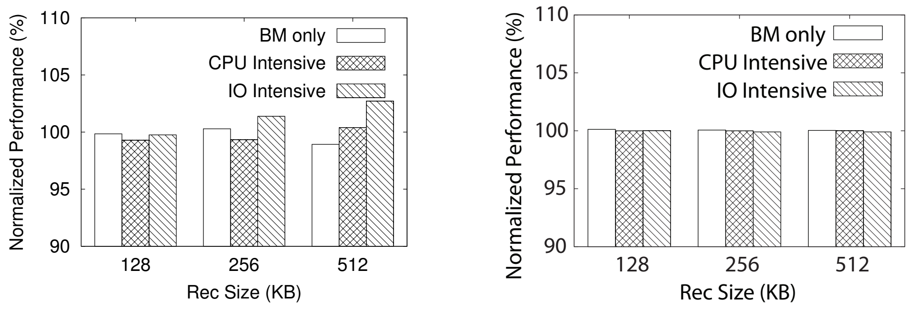
Fig. 16. Intel X25M G1: Read Performance on Filebench and Postmark

Fig. 16(a) illustrates the performance improvement obtained by SmartCon for file server, multi-stream read, single stream read and mail server workloads. For the first three workloads, there is virtually no performance gain. This is because of the large IO size of these workloads. On the other hand, SmartCon yields 5.5% performance gain in mail server workload. This is because mail server generates mostly small size IO (16

KByte) which SmartCon can exploit busy-waiting. As shown in Fig. 16(b), SmartCon yields 11% and 9% improvement in Postmark for read and write, respectively.

7.5. SmartCon for large IO

It is important to make sure that SmartCon performs as well as context switch based IO subsystem does for large IO. Fig. 17 shows the performance of large IO with IOZONE, and record sizes of 128KB, 256 KB, and 512 KB for two storage device (RevoDrive and Intel X25M G1). SmartCon performs on par with mainstream IO subsystem.



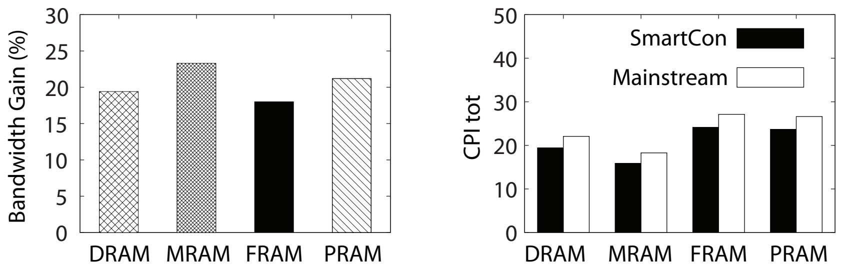
(a) OCZ RevoDrive3 (b) Intel X25M G1

Fig. 17. SmartCon Overhead from 128KB to 512KB IO size (NP 100% in Y axis denotes I/O perf. of Mainstream)

7.6. Performance Projections For Various NVRAM Technologies

In this section, we provide some projections of SmartCon performance for four different NVRAM technologies: DDR (50 ns), PCRAM (70 ns), FRAM (68 nsec) and MRAM (35 ns) using our simple performance projection model introduced in section 6 and using the calibration parameters listed in Table III.

Fig. 18 illustrates the effective CPI values, and performance ratios for DRAM, FRAM, PRAM and MRAM devices, respectively. SmartCon is expected to bring 80% performance gain against mainstream context switch based IO.



(a) Bandwidth Gain for Read (b) CPI

Fig. 18. Performance Gain of SmartCon under Different Devices (CPI_{tot}: CPI in Eq. 1)

7.7. Impact of SmartCon on Cache Pollution

The reduction of context switches by SmartCon could reduce cache pollution and hence lead to better performance. We examine the cache miss behaviors with and without SmartCon, respectively. As discussed earlier, only the indirect overhead is significant. To quantify this indirect overhead of context switches, we ran IOZone on RAMDISK along with four other processes, each of which accesses files on the hard disk drive. We measured the Instruction TLB misses, Data TLB misses, L1 and L2 cache miss rates with varying number of IO sizes. The file size used in this experiment was 256MB. The cache performance is assessed from the hardware performance counters.

Fig. 19 shows the miss ratio of different caches as a function of IO size: Instruction TLB, Data TLB, L1 Data Cache, L1 Instruction Cache, L2 Data Cache and L2 Instruction Cache. Fig. 19(a) shows that the Instruction TLB miss rate is too small ($< 1\%$) to affect the performance. For larger IO sizes of 16 KB, 128 KB and 512 KB, the Inst TLB miss rate becomes lower because of fewer instructions per byte of data transferred.

The Data TLB miss rate remains quite high for both SmartCon IO and mainstream IO, as shown in Fig. 19(b); however, the SmartCon value is lower in all IO sizes. On the average, Data TLB miss rate decreases by 7% by using SmartCon. The reason for Data TLB miss rate being much larger than Instruction TLB miss rate is that IO intensive workloads generate significant data traffic but have a small instruction footprint. Overall, for 4 KB IO size, SmartCon can reduce the Instruction TLB misses by 15% and data TLB misses by 16% or more.

Fig. 19(c) and 19(d) show the L1 Instruction and Data Cache miss rates, respectively. We observe that the miss rate differs by a factor of 20 between Instruction and Data TLB. The miss rate of L1 Data Instruction caches are relatively similar. For 4 KB record size, L1 instruction cache miss rate decreases from 13.2% to 8.1% by using SmartCon. For 16 KB record size, SmartCon provides better improvement than Mainstream IO. For 128KB and 512KB, the instruction L1 miss rate becomes very small for essentially the same reasons as given for instruction TLB miss rate. For 4 KB record size, SmartCon reduces the L1 Data Cache miss rate from 13% to 4.3%. For 16 KB and 128 KB IO size, the improvement provided by SmartCon is significantly smaller.

Because of the large size of L2 cache, the L2 miss rate is only about 1/200th of L1 miss rate. Figs. 19(e) and 19(f) show L2 cache miss rate caused by instruction fetches and data `loads`, respectively. Unlike instruction TLB and L1, where the miss rate decreases with IO size, L2 instruction miss rate generally increases with IO size. This behavior has to do with effectiveness of L2 caching. A large IO size makes it more difficult to contain the working set in L2 cache and the miss rate increases. This is particularly obvious for 512KB IO size since the L2 size is only 4MB. As for the impact of SmartCon, the L2 instruction miss rate is somewhat higher but the data miss rate is significantly lower. In particular, SmartCon improves L2 cache miss rate by upto 77%.

7.8. Impact of `PAUSE` instruction on SmartCon

In this section, we introduce `PAUSE` instruction [Orenstien and Ronen 2004] and apply it to SmartCon. The `PAUSE` instruction gives a hint to physical processor that this loop is just a spin-wait loop so that the physical processor which provides hyper-threading can switch to another logical processor in order to increase system throughput. The benefit of `PAUSE` instruction with hyper-threading is two-fold: (i) switching other logical processor and (ii) reducing power consumption by using ”nop”.

We tested the `PAUSE` enabled SmartCon with Intel X25M by inserting `PAUSE` instruction on spin-wait loop for checking the IO completion. We ran IOzone with 4KB and 8KB IOsize. Fig. 20 illustrates the time series of CPU utilization under SmartCon

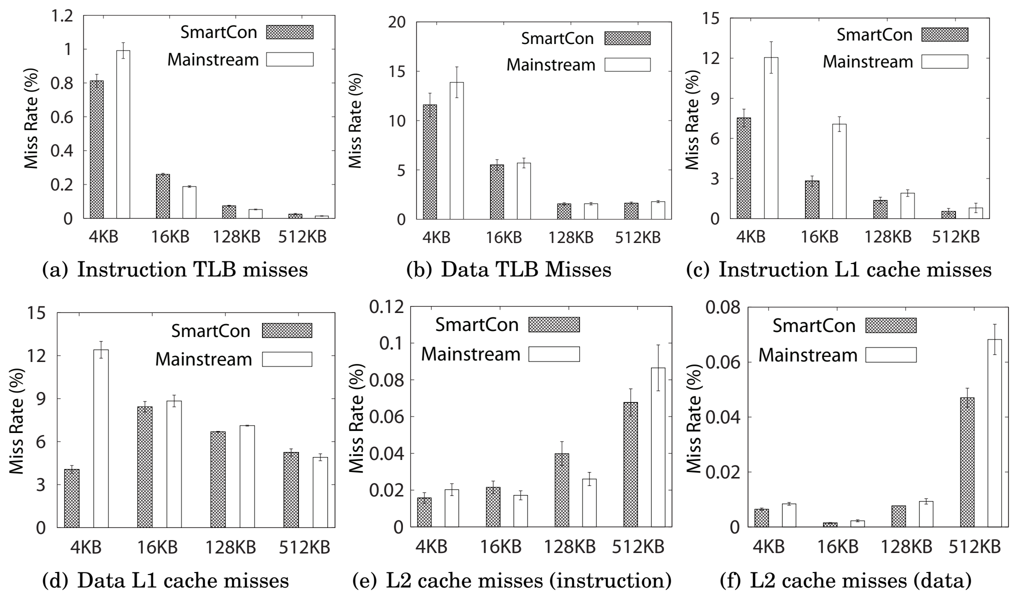


Fig. 19. L1, L2 and TLB Behavior in Mainstream and SmartCon enabled OS

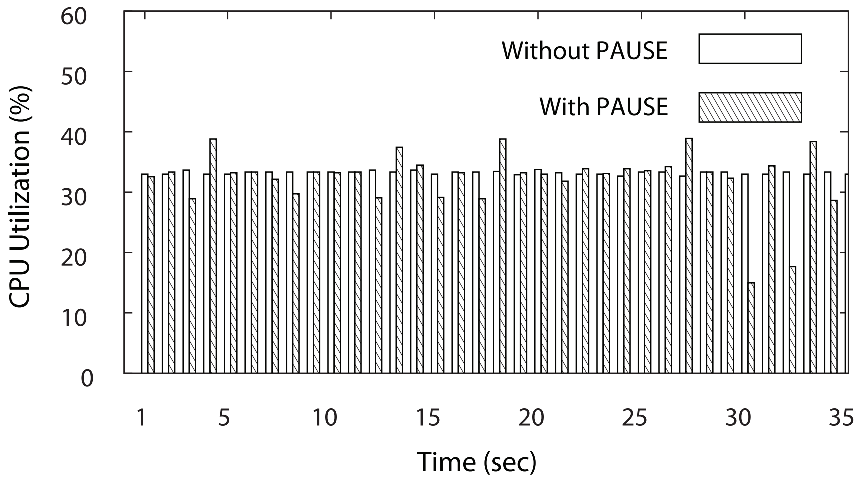


Fig. 20. With/without `PAUSE` Instruction on SmartCon

based IO. The white bar and the comb-pattern bar denotes the result without and with `PAUSE` instruction, respectively. The CPU utilization and the IO performance of SmartCon with and without `PAUSE` instructions are similar. With `PAUSE` instruction, the CPU utilization varies more widely than without `PAUSE` instruction.

8. CONCLUSIONS AND FUTURE WORK

Recent advancement in storage technologies have produced a new set of storage devices that are orders of magnitude faster than mainstream hard disk. They include the already mature NAND flash based SSD's and forthcoming byte-addressable NVRAM's, e.g. STT-MRAM and PCRAM. Also, recent advancements in multicore CPUs makes

CPU cycles generally plentiful. For many workloads, the performance of the storage subsystem is one of the key factors that governs the overall system performance.

In this work, we argue that context switch based IO subsystem leaves much to be desired in fully exploiting the state of art storage devices and multicore CPUs. We propose a new mechanism called SmartCon (Smart Context Switch) that conditions context switch on the performance characteristics of the IO device, the system load, and the IO request size.

We prototyped `SMARTCON` in Linux Operating System (2.6) and performed extensive experiments to verify the performance benefits under a wide variety of system loads. SmartCon was tested with three representative storage devices: RAMDISK, high-end SSD, and low-end SSD. SmartCon yields upto 45% improvement in IO performance and decreases involuntary context switches by upto 84%. We verify that SmartCon does not negatively affect the overall system throughput. We also introduce a simple analytic model to project performance advantage of SmartCon for future NVRAM technologies.

In the future, we plan to address the energy consumption of SmartCon because the lack of context switch stalls the CPU and an intelligent mechanism to use available sleep stacks (e.g. C6) becomes essential to reduce energy waste. We will also examine the impact of device cache on SmartCon performance and devise mechanisms to include it in the decision algorithm, if necessary.

REFERENCES

Anant Agarwal, John Hennessy, and Mark Horowitz. 1988. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.* 6, 4 (1988), 393–431.

M.R. Ahmadi and D. Maleki. 2010. Performance evaluation of server virtualization in data center applications. In *Proc. of International Symposium on Telecommunications (IST)*. IEEE, Tehran, Iran.

Ameen Akel, Adrian M. Caufield, Todor l. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. Onyx: A Prototype Phase Change Memory Storage Array. In *Proc. of USENX HotStorage*. Portland, OR.

Suparna Bhattacharya, John Tran, Mike Sullivan, and Chris Mason. 2004. Linux AIO Performance and Robustness for Enterprise Workloads. In *Proc. of Linux Symposium*. Ottawa, Canada.

G.W. Burr, B.N. Kurdi, J.C. Scott, C.H. Lam, K. Gopalakrishnan, and R.S. Shenoy. 2008. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development* 52, 4-5 (2008), 449–464.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 385–395. DOI:http://dx.doi.org/10.1109/MICRO.2010.33

Feng Chen, Michael P Mesnier, and Scott Hahn. 2014. A Protected Block Device for Persistent Memory. In *Proceedings of The 30th International Conference on Massive Storage Systems and Technology (MSST'14)*.

Joel Coburn, Adrian M. Caufield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-volatile Memories. In *Proc. of the international conference on Architectural support for programming languages and operating systems (ASPLOS)*. Newport Beach, CA.

J. Corbet, A. Rubini, and G. Kroah-Hartman. Feb. 2005. *Linux device drivers*. O'Reilly Media, Inc.

Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. 2007. Context switch overheads for Linux on ARM platforms. In *Proc. of workshop on Experimental computer science (ExpCS)*. San Diego, CA, USA.

Samsung Electronics. June 2005. *OneNAND Specification ver.1.2*.

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox. 2004. Lazy Asynchronous I/O for Event-Driven Servers. In *Proc. of USENIX Annual Technical Conference (ATC)*. Boston, MA.

R. F. Freitas and W. W. Wilcke. 2008. Storage Class Memory: The next storage system technology. *IBM Journal of Research and Development* 52, 4/5 (2008), 439–447.

FusionIO. 2010. ioDrive Product Family User Guide - Linux for Driver Release 2.1.0. (Aug. 2010).

HenkPoley. Feb, 2014. A Look Back at Single-Threaded CPU Performance. http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/. (Feb, 2014).

M. Hosomi, H. Yamagishi, and et. al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *Proc. of IEEE Intl. Electron Devices Meeting*. Washington, DC, UA, 459–462.

A Huffman. 2012. Nvm express, revision 1.0 c. *Intel Corporation* (2012).

Intel. 2011. Light Peak Technology. *Web site: http://en.wikipedia.org/wiki/Light_Peak* (Jan. 2011).

E. Ipek, J. Condit, B. Lee, E.B. Nightingale, D. Burger, C. Frost, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS symposium on Operating systems principles (SOSP)*. Big Sky, MT, USA.

Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 309–320. https://www.usenix.org/conference/atc13/technical-sessions/presentation/jeong

Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. 2009. A new look at the roles of spinning and blocking. In *Proc. of the International Workshop on Data Management on New Hardware (DaMoN)*. ACM, New York, NY, USA, 21–26.

M. Tim Jones. 2006. Linux initial RAM disk (initrd) overview. *Web site: https://www.ibm.com/developerworks/library/l-initrd* (July 2006).

J. Jung, J. Choi, Y. Won, and S. Kang. 2009. Shadow Block: Imposing Block Device Abstraction on Storage Class Memory. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*. Grenoble, France.

J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. 2010. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Trans. on Storage* 6, 1 (2010), 1–25.

K. Kant. 2008. Exploiting NVRAM for Building Multi-Level Memory Systems. In *Proc. of Intl. Workshop on OS Technologies for Large Scale NVRAM (NVRAMOS): Presentation*. Jeju, Korea.

K. Kant and Y. Won. 1999. Server capacity planning for Web traffic workload. *Knowledge and Data Engineering, IEEE Transactions on* 11, 5 (1999), 731 –747.

Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. 1991. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proc. of the ACM symposium on Operating systems principles (SOSP)*. ACM, New York, NY, USA, 41–55.

Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 406–422.

J. Katcher. 1997. Postmark: A new file system benchmark. (1997).

M. Kobayashi. 1986. An empirical study of task switching locality in MVS. *IEEE Trans. on Computers* 100, 35 (Aug. 1986), 720–731.

Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proc. of the annual international symposium on computer architecture (ISCA)*. ACM, New York, NY, USA.

Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proc. of workshop on Experimental computer science (ExpCS '07)*. San Diego, CA, USA.

Jing Li, Charles Augustine, Sayeef Salahuddin, and Kaushik Roy. 2008. Modeling of failure probability and statistical design of spin-torque transfer magnetic random access memory (STT-MRAM) array for yield enhancement. In *Proc. of Design Automation Conference (DAC)*. Anaheim, CA, UA, 278–283.

Fang Liu, Fei Guo, Yan Solihin, Seongbeom Kim, and Abdulaziz Eker. Characterizing and modeling the behavior of context switch misses. In *Proc. of Intl. conf. on Parallel architectures and compilation techniques (PACT)*. 91–101.

R. McDougall and J. Mauro. 2005. FileBench. (2005).

L. McVoy and C. Staelin. 1996. LMBENCH: Portable tools for performance analysis. In *Proc. of the USENIX Annual Technical Conference (ATC)*. San Diego, CA, USA, 279–294.

Microsoft corp. msdn. 2010. Ramdisk. *Web site: msdn.microsoft.com/en-us/library/ff544551(VS.85).aspx* (Aug. 2010).

Kevin OBrien. May, 2013. OCZ RevoDrive 3 X2 480GB Review. (May, 2013).

Doron Orenstien and Ronny Ronen. 2004. Low-power processor hint, such as from a PAUSE instruction. (Feb. 3 2004). US Patent 6,687,838.

J.K. Ousterhout. 1990. Why are not operating systems getting faster as fast as hardware. In *Proc. of the Summer 1990 USENIX Conf.* Anaheim, CA, 247–256.

S. Park and K. Shen. 2009. A performance evaluation of scientific i/o workloads on flash-based ssds. In *Proc. of Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS)*. New Orleans, LA.

Moinuddin K. Qureshi, Michele M. Franceschini, Luis A. Lastras-Montaño, and John P. Karidis. 2010. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *Proc. of the annual international symposium on computer architecture (ISCA)*. ACM, New York, NY, USA.

Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Revers. 2009. Scalable High Performance Main Memory System Using Phase-Change Memory Technolohy. In *Proc. of ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Austin, Texas, US, 24–33.

K. Salah and A. Qahtan. 2009. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Computer Communications* 32, 1 (2009), 179 – 188.

Samsung Electronics. June 2007. 1Gb C-die DDR3 SDRAM Specification. (June 2007).

Patrick Schmid and Achim Roos. Sep, 2008. Intel X25-M Solid State Drive Reviewed. http://www.tomshardware.com/reviews/Intel-x25-m-SSD,2012.html. (Sep, 2008).

Johan Starner and Lars Asplund. Measuring the cache interference cost in preemptive real-time systems. In *Proc. of ACM SIGPLAN/SIGBED conf. on Languages, compilers, and tools for embedded systems (LCTES)*. 146–154.

D. Tam, R. Azimi, L. Soares, and M. Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. San Diego, CA.

The Open Group Base Specifications Issues 6 IEEE Std 1003.1. 2003. http://www.opengroup.org/onlinepubs/007904975. (2003).

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for non-Volatile Byte-Addressable Memory. In *Proc. of the Usenix Conference on File and Storage Technologies (FAST)*. San Jose, CA.

G. Venkatasubramanian, R.J. Figueiredo, R. Illikkal, and D. Newell. Sao Paulo, Brazil, Oct. 2009. TMT-A TLB Tag Management Framework for Virtualized Platforms. In *Proc. of 21st Intl Symp on Computer Arch and High Perf Computing*. 153–160.

Haris Volos, Nadres Jann Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proc. of the international conference on Architectural support for programming languages and operating systems (ASPLOS)*. Newport Beach, CA.

R.P. Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM* 27, 10 (1984), 1013–1030.

A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser. 2003. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *Proc. of the Eighth Asia-Pacific Computer Systems Architecture Conference (ACSAC)*. Aizu-Wakamatsu, Japan.

Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. 2009. Hybrid cache architecture with disparate memory technologies. In *Proc. of the annual international symposium on computer architecture (ISCA)*. ACM, New York, NY, USA.

S. Yamada and S. Kusakabe. 2008. Effect of context aware scheduler on TLB. In *Proc. of IEEE Intl. symp. on Parallel and Distributed Processing*. Miami, Florida, USA, 1–8.

J. Yan and W. Zhang. 2008. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symp (RTAS)*. St. Louis, MO, USA, 80–89.

Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt.. In *Proceedings of the 10th USENIX Conference on FAST (FAST'12)*. 3.

Received May 2013; revised May 2013; accepted May 2013