Truly Non-blocking Writes

Luis Useche[†] *Ricardo Koller*[†] *Raju Rangaswami*[†] *Akshat Verma*[*]
Florida International University[†] IBM Research - India[*]

Abstract

Writing data within user or operating system memory often encounters the classic *read-before-write* problem whereby the page written to must first be read from the backing store, effectively blocking the writing process before modifications are made. Unfortunately, the large gap between memory and storage access performance adversely affects workloads that require substantial *read-before-write* operations when accessing memory. In this paper, we present techniques that make writes to memory truly non-blocking. The basic approach involves absorbing writes immediately in temporary buffer pages and asynchronously merging the updates after reading in the on-disk version of the page. Doing so improves system performance by first, reducing blocking of processes and second, improving the parallelism of data retrieval from the backing store leading to better throughput for *read-before-write* operations. We analyze the potential benefits of our approach using full-system memory access traces for several benchmark workloads and present techniques that commodity operating systems can employ to implement non-blocking writes.

1 Introduction

Writing data to main memory is extremely fast. Well, ... most of the time. When using demand-paged virtual memory and file systems, the main memory caches a *subset* of the data contained within a backing store, typically a hard disk or SSD (array) device. However, memory references are done at a much smaller granularity (32 or 64 bit words) than is possible in backing stores. When data not present in memory is read or modified by the process, the operating system (OS) must fetch an entire page, typically 4KB or 8KB in commodity OSes, that contain the few bytes referenced by the processor. *Page fetches* occur in two scenarios: (1) a page mapped to the process address space (anonymous or file system page cache page) not resident in physical memory gets accessed by a machine instruction (e.g., `LOAD` or `STORE`) causing a page fault, or (2) a system call accesses the OS page cache (e.g. `OPEN`, `READ`, or `WRITE`). Since a page fetch blocks the process, the performance of the running process during that time is limited by the performance of the backing store.

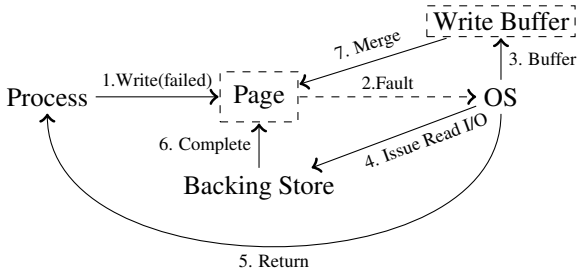

Figure 1: A non-blocking write in action. *The first step, a write reference, fails because the page is not yet in memory. The dashed boxes are non-active entities.*

For read references, such blocking cannot be avoided since there is no other provision to correctly generate the data being read. Interestingly, the same blocking approach has been applied to handle write references in commodity operating systems and hypervisors till today (e.g., see recent Linux 2.6.34.5 and Xen 4.1.0 kernels). These include write references to swap-backed process memory or disk-backed file system pages, either directly (for anonymous and memory mapped pages) or via system calls (for OS cached pages). Thus, the writing process blocks in case the page being referenced is not in core memory until the referenced page is synchronously read from the backing store, leading to a *read-before-write* requirement [11]. This paper demonstrates that writes *can* and *should* be handled differently and proposes an approach to eliminate blocking for all write references to memory. We observe that in case of write references, instead of blocking the process to read in the page, the operating system can absorb such writes in temporary buffer pages and allow the process to continue executing immediately. The operating system can issue the page read I/O to the backing store asynchronously and merge the update later when the page has been read into memory. A graphical representation of this process ap-

pears in Figure 1.

The proposed approach improves system performance in two ways. First, it immediately unblocks the process which is then free to execute subsequent instructions and make progress; the originally blocking page fetch operation can now overlap with useful computation. Second, it improves the parallelism of data retrieval from the backing store by creating the ability to keep many outstanding I/O (OIO) operations to handle multiple page fetches by a single process at the same time. Doing so leads to better throughput for both SSD and hard disk based backing stores, both of which offer better I/O throughput with multiple outstanding I/O operations. For example, we found an almost 5x increase in IOPS for a PCIe OCZ Revodrive 160GB SSD when the number of OIO were increased from 1 to 16.

In the next section, we motivate the need for non-blocking writes in commodity systems. We then present our approach to make all memory write references non-blocking (§3). We quantify the potential benefits of our approach using full system memory access traces for a variety of workloads (§4). Finally, we summarize our work and point to future directions.

2 Motivating Non-blocking Writes

In this section, we identify technology trends and workload behaviors that motivate non-blocking writes.

2.1 Trends in Page Fault Rates

Two trends point to the likelihood of increasing page fault rates in systems. First, multi-core systems and virtualization are changing memory footprints fundamentally. Higher application tenancy ratios and corresponding increase in in-memory working sets demand ever-increasing amounts of DRAM, a resource that incurs high cost and energy consumption. A second trend that is now evident is that the memory hierarchy is also changing fundamentally. High-performance flash-based SSDs are becoming mainstream as backing store devices and now vendors are introducing SSDs in servers as well [5, 6]. Researchers have already begun designing hybrid memory systems and flash-based virtual memory and file system caching systems that are motivated by these trends [13, 9]. Thus, we can expect an increasing off-loading of the traditionally DRAM-resident working set data to this additional high-performance SSD layer in the storage stack, indicating a move to higher page fault rates in future systems.

2.2 *read-before-write*: For Real?

How common is *read-before-write* in reality? Intuitively, *read-before-write* occurs when a page in the backing store is overwritten. Here, it is important to differentiate *over-writes* from *allocation-writes*. While the former modifies an existing page incurring a *read-before-write* operation, the latter allocates a new page on the fly to write into and is thus free from *read-before-write*.

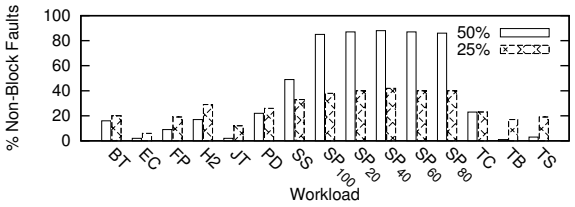

Figure 2: Fraction of page faults that benefit from non-blocking writes for various workloads with two memory sizes: 50% and 25% of each workload's footprint.

As mentioned before, page over-writes causing *read-before-write* can occur for both virtual memory pages and file system pages. To estimate the occurrence of *read-before-write* in real workloads, we analyzed full system memory access traces for a heterogeneous set of workloads including server, image processing, and developer workloads. These workloads are summarized in Table 1.

We built a memory simulator that reports the number of page faults, the number of write faults that incur *read-before-write* operations, and the number of write faults that can truly benefit from non-blocking writes. The distinction between the latter two lies in the consideration that some non-blocking writes of *read-before-write* operations are quickly followed by a blocking read fault to the same page, thereby eliminating any potential benefit due to the non-blocking write. We provide more details on how this data was collected and analyzed, and the simulator we built in Section 4.

Figure 2 shows that with a DRAM provisioned for storing 50% of the the total memory pages referenced, there is a substantial fraction – as much as 80% – of the total page faults (including both read and write faults) that can benefit from non-blocking writes for a variety of workloads.

2.3 Alternatives to Non-blocking Writes

To the best of our knowledge, we are not aware of any prior work that explicitly identifies or addresses the *read-before-write* problem. We now consider alternate optimizations that can possibly achieve the two goals of non-blocking writes, i.e., reduce blocking of process execution and increase parallelism of access to the backing store.

One approach to reducing the blocking of processes is provisioning adequate DRAM to eliminate write page faults. However, for both process memory and file system writes, the footprint of a workload is unpredictable and potentially unbounded. Moreover, as discussed earlier, technology trends do not support this as a viable solution; faster devices that serve as more cost effective

and energy-efficient replacement for DRAM to store relatively cold data are becoming available.

Prefetching [14] is an alternative approach that can reduce blocking by anticipating memory accesses and prefetching necessary pages to eliminate page faults. Moreover, it can also increase the efficiency of access to the storage back-end by issuing multiple page prefetch requests simultaneously. Prefetching has been demonstrated as valuable when using an SSD-backed virtual memory system [13]. Unfortunately, prefetching can incur both false positive and false negative fetches that pollute memory. With non-blocking writes, pages are only brought to memory when they are accessed. Fortunately, the two methods are not exclusive and can be used in conjunction.

3 Solution Approach

Page over-writes can be either supervised or unsupervised by the OS. Supervised page over-writes occur via system calls, typically to file system data and metadata pages. Un-supervised page over-writes occur in user pages and memory mapped file pages.

For supervised over-write to an out-of-core page, the OS has all the information it needs to set up the non-blocking write. Once the application makes an OS page-modifying system call, the OS can allocate a buffer to temporarily store the data. It can then use the offset and size information from the system call parameters to set up the asynchronous read of the relevant file pages and atomically merge these with the saved data after the read is completed. In case a non-blocking write to the over-written page is already in progress, the new data is stored in the same buffer to be merged when the in-progress asynchronous read completes.

For an un-supervised over-write to an out-of-core page, the processor would generate a page fault. Commodity processors provide the reference address and the instruction that generated the page fault to the OS. Unfortunately, the number of bytes accessed by the instruction generating the page fault is usually not available. Modern processor architectures (e.g., x86) provide complex instructions that can perform multi-word writes in the same instruction [7]. In the rest of the section, we present three approaches to implement non-blocking writes in case of un-supervised page over-writes.

3.1 Full Feature Hardware

This solution assumes that the processor provides all the information needed to implement non-blocking writes when interrupted due to a page fault. With such hardware, the OS can use the virtual address as well as the number of bytes accessed on a page fault and proceed similarly to the solution for the supervised over-write. In the future, processor manufacturers could augment CPU page fault interrupts with such information if not available. The advantage of this approach is that the hardware provides all the information needed and no additional time needs to be spent by the OS to determine such information. If such support is not universal, the solution described above becomes architecture dependent.

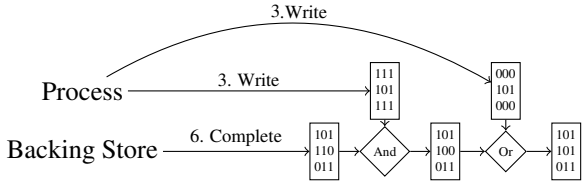

Figure 3: Non-blocking writes via the *page diff-merge technique*. *The step numbers match the ones in Figure 1. Rectangles refer to pages in memory.*

3.2 Opcode Disassembly

A second solution is to disassemble the opcode of the instruction that generated the page fault to extract the number of bytes written to the page. Opcode disassembly functionality for several commodity ISAs is implemented in several important systems software projects that need to perform binary analysis or translation such as QEMU [2], VMware[15], and the Linux kernel. Although these implementations are highly optimized, disassembly would still increase the computation demand in the time-critical page fault handler. This solution can be implemented for any machine architecture.

3.3 Page Diff-Merge

Our last solution uses a *page diff-merge* technique as illustrated in Figure 3. On a write fault, we write the modifications in two temporary pages using the available single-step feature on modern processors: a *0-page* initialized with all zero bits and an *1-page* initialized with all ones. With these two pages available, the OS can construct the new version of the page read asynchronously using a merge technique that performs bitwise *or* and *and* with the 0-page and 1-page respectively. This approach is architecture-independent and no special hardware is necessary. Unfortunately, it adds even more time and space overhead than opcode disassembly because of the additional data operations and page faults incurred, one each for writing to the 0-page and 1-page.

4 Quantifying Benefits

In this section, we quantify the potential benefits of non-blocking writes using several workloads.

4.1 Full System Memory Traces

We modified the QEMU machine emulator [2] to collect full system memory traces for the x86 architecture. We modified the software-MMU version of QEMU and inserted tracing code at binary translation points. Specifically, each time a *translation block* (i.e., binary blocks

Workload	ID	Footprint (MB)	Exec. Time (secs)	# References ($\times 10^6$)
batik	BT	149	17.60	25
eclipse	EC	223	7.20	11
fop	FP	144	11.16	32
h2	H2	722	44.19	386
jython	JT	540	49.68	128
pmd	PD	170	20.86	60
tomcat	TC	215	33.39	118
tradebeans	TB	337	23.84	87
tradesoap	TS	335	31.56	84
postmark	SS	256	9.9	69
specpower-20	SP_{20}	218	22	44
specpower-40	SP_{40}	224	22	48
specpower-60	SP_{60}	214	22	53
specpower-80	SP_{80}	214	22	60
specpower-100	SP_{100}	211	22	63

Table 1: Workloads include a mix of DaCapo benchmark suite 9.12 [3], PostMark [8], and SPECpower [10]. *SP_X X indicates the percentage of load in the system. PM is set to the small-small configuration used by Riska et al.[12].*

between jumps and jump return points) is sent to the inline compiler, we insert code for recording each load and store instruction. Since the addresses used by loads and stores may be unknown at translation block compile time, the appropriate register values are recorded at run time. Guest virtual address to guest physical address translation is obtained via a custom extension to the software TLB managed by QEMU.

The memory accesses go through a cache simulator (code borrowed from $valgrind$ [1]) at runtime. The caches used were a 64KB I1 instruction cache, a 64KB D1 data cache and a 4MB L2 data cache, all using lines of 64 bytes. Timestamps were assigned by mapping instruction counts to wall-clock time based on execution times observed in real hardware. We ran Linux on the modified QEMU emulator configured with 1GB of physical memory and trace the workloads detailed in Table 1. The emulator itself ran on a 2.93 GHz Intel Xeon X7350 processor. The traces we collected included both user-level and kernel memory references and included the following information: *emulator time-stamp, instruction-count, process CR3, virtual address, physical address, access-mode (R/W), machine-mode (kernel/user), page access privileges.*

4.2 Virtual Memory Simulation

We built a virtual memory simulator that given a memory size and a memory trace, simulates hits, misses, and evictions of memory pages. On every memory reference, it reports the timestamp, operation mode (read or write), and event (hit, miss, or evict). More importantly, the simulator is designed to report the number of write faults that can benefit from non-blocking writes. To do so, the simulator must *(i)* be able to distinguish *over-writes* from *allocation-writes*, and *(ii)* determine which write faults can really benefit from non-blocking writes.

As discussed previously, allocation-writes do not trigger I/O operations while over-writes may. Unfortunately, we do not have sufficient information in our traces to entirely distinguish one from the other. In order to minimize the occurrence of false positives when detecting over-writes, we use two heuristics. First, we consider the first write access to every page in the trace conservatively as an allocation write by default. Second, we use both the virtual and physical addresses to uniquely identify a page, instead of just the virtual address or the physical address alone. This eliminates false positives in detecting overwrites when the same virtual address is reused to map to a different physical address or vice-versa.

Finally, the simulator also employs a model to predict I/O latency for various values of OIO which is then used to determine when an asynchronous read related to a non-blocking write would complete. Using an approximation proposed by Gulati *et al.* that latency varies linearly with OIO [4] and training this model on a few points with a real SSD (PCIe OCZ Revodrive 160GB), we were able to predict the latency of the device for any arbitrary OIO value.

4.3 Fraction of Non-blocking Write Faults

We measured the fraction of page faults that benefit from non-blocking writes for all workloads in Table 1. Figure 2 shows these results. When the provisioned DRAM is half of the total memory footprint (total size of memory referenced) for each workload, 2-88% of the faults are write faults that benefit from non-blocking writes. When the provisioned DRAM is reduced to a fourth of the memory footprints, the fraction of non-blocking write faults is 7-42%. This implies that there is substantial potential for improving the overall performance of page fault related work for many of these workloads. For some workloads, a reduction in memory size causes a reduction in the percentage of non-blocking writes. For these workloads, higher page fault rates (as reduced memory sizes) lead to more of the pages involved in non-blocking writes being selected for eviction. Our simulator correctly revokes the non-blocking write status for such pages given that they must now block for the completion of the background read before they can be evicted.

4.4 Outstanding Write Faults

While the fraction of non-blocking write faults are partially indicative, they do not suggest how the absolute number of simultaneous non-blocking writes vary over time. To address this consideration, we define the *outstanding write faults* (OWF) as the number of write faults that can still benefit from non-blocking writes at any time during the execution of a process. OWF gets incremented each time there is a write to an out-of-core page. An asynchronous read to the page is also initiated at that time

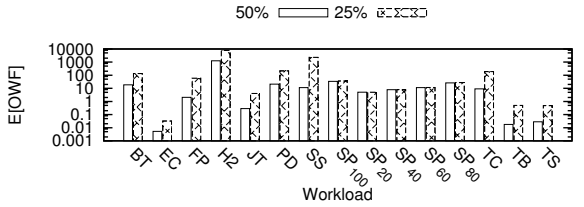

Figure 4: Expected OWF for various workloads with two different memory sizes: 50% and 25% of workload footprint.

and the page is marked as *in-io*. When there is a read reference by the process to an *in-io* page, or an *in-io* page is evicted, or if the simulated asynchronous read due to an *in-io* page completes, the OWF value gets decremented.

Figure 4 reports the expected value (time-weighted average) of the OWF for each of the workloads when the DRAM size is configured to be 50% or 25% of the total memory footprint. First, we note that all workloads show values greater than zero, indicating that each can benefit from non-blocking writes. For most workloads, the OWF ranges from 2 to 33, indicating a healthy opportunity for parallelizing the asynchronous reads due to non-blocking writes. The two exceptions are EC, which has the lowest OWF value (0.005), and H2, which has an exceptionally high OWF value (1259.07).

4.5 Estimating Overall Savings

To estimate how non-blocking writes would impact execution times, we revisit the percentage of page faults that can benefit from non-blocking writes (discussed earlier in Figure 2). This gives an upper bound on the savings in running time that our set of benchmarks could have with non-blocking writes. With 50% DRAM provisioning relative to memory footprint size, half of the workloads show 20% or less page faults that benefit from non-blocking writes. However, the range is 30% to 80% for the remaining half which incur more paging activity. Combining this finding with those from previous studies which show that applications using paging and are heavily optimized spend more than 40% on disk I/O [13], we could estimate an overall reduction of 12% to 32% in application execution times for these workloads.

4.6 Overhead

Non-blocking writes are not free from overhead. The temporary buffer pages that hold modifications while the asynchronous read is in progress consume additional memory resources. The average memory overhead of the system is proportional to the $E[\text{OWF}]$ of the application since the system must buffer data for each outstanding write fault. This number, as shown in Figure 4, is usually low. On average, we see an $E[\text{OWF}]$ close to 10. If we assume a conservative 8KB (two pages) for each OWF as per the page diff-merge technique, we would use 80 KB of memory, representing a nominal overhead. Our worst case, H2, has an $E[\text{OWF}]$ of less than 1300, which results in roughly 10MB of overhead using the same calculation as above.

5 Conclusions and Future Work

We presented *non-blocking writes*, a technique that eliminates the blocking *read-before-write* requirement incurred in handling write page faults. Non-blocking writes absorb modifications to out-of-core pages in temporary buffer pages allowing the process to continue immediately, while reading-in the page and merging changes later asynchronously. We quantified the benefits of non-blocking writes and estimated a savings in execution time of 12-32% across a variety of workloads. The natural next step for our work will involve an actual operating system implementation of non-blocking writes to accurately quantify its benefits for real workloads.

References

[1] Valgrind. `http://valgrind.org/`.

[2] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX ATC* (April 2005).

[3] BLACKBURN, S. M. ET AL. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of OOPSLA* (New York, NY, USA, Oct. 2006), ACM Press, pp. 169–190.

[4] GULATI, A., KUMAR, C., AHMAD, I., AND KUMAR, K. Basil: Automated io load balancing across storage devices. In *FAST* (2010), pp. 169–182.

[5] HP. Solid state storage technology for Proliant servers. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c01580706/c01580706.pdf`.

[6] INFOWORLD ARTICLE. Sun pushes SSDs in servers to reduce costs, improve performance. `http://www.infoworld.com/t/hardware/sun-pushes-ssds-in-servers\-reduce-costs-improve-performance-980`.

[7] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developers Manual. Specification, 2007.

[8] KATCHER, J. PostMark: A New File System Benchmark. *Technical Report TR3022. Network Appliance Inc.* (October 1997).

[9] KGIL, T., AND MUDGE, T. Flashcache: a nand flash memory file cache for low power web servers. In *Proc. of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (October 2006).

[10] LANGE, K.-D. Identifying shades of green: the specpower benchmarks. *computer 42* (2009), 95–97.

[11] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996, pp. 163, 196.

[12] RISKA, A., LARKBY-LAHET, J., AND RIEDEL, E. Evaluating block-level optimization through the io path. In *Proc. of the USENIX ATC* (June 2007).

[13] SAXENA, M., AND SWIFT, M. M. FlashVM: Revisiting the Virtual Memory Hierarchy. In *Proc. of USENIX ATC* (June 2010).

[14] SHRIVER, E., SMALL, C., AND SMITH, K. A. Why does file system prefetching work? In *Proc. of USENIX ATC* (1999).

[15] VMWARE INC. VMWare ESX Server User's Manual Version 1.5. `http://www.vmware.com/support/pubs/esx_pubs.html`, 2002.