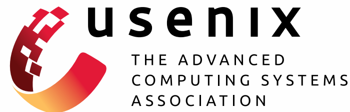

I/O Speculation for the Microsecond Era

Michael Wei, *University of California, San Diego;* Matias Bjørling and Philippe Bonnet, *IT University of Copenhagen;* Steven Swanson, *University of California, San Diego*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/wei

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

I/O Speculation for the Microsecond Era

Michael Wei[†], Matias Bjørling[‡], Philippe Bonnet[‡], Steven Swanson[†]
[†]*University of California, San Diego* [‡]*IT University of Copenhagen*

Abstract

Microsecond latencies and access times will soon dominate most datacenter I/O workloads, thanks to improvements in both storage and networking technologies. Current techniques for dealing with I/O latency are targeted for either very fast (nanosecond) or slow (millisecond) devices. These techniques are suboptimal for microsecond devices - they either block the processor for tens of microseconds or yield the processor only to be ready again microseconds later. Speculation is an alternative technique that resolves the issues of yielding and blocking by enabling an application to continue running until the application produces an externally visible side effect. State-of-the-art techniques for speculating on I/O requests involve checkpointing, which can take up to a millisecond, squandering any of the performance benefits microsecond scale devices have to offer. In this paper, we survey how speculation can address the challenges that microsecond scale devices will bring. We measure applications for the potential benefit to be gained from speculation and examine several classes of speculation techniques. In addition, we propose two new techniques, hardware checkpoint and checkpoint-free speculation. Our exploration suggests that speculation will enable systems to extract the maximum performance of I/O devices in the microsecond era.

1 Introduction

We are at the dawn of the *microsecond era*: current state-of-the-art NAND-based Solid State Disks (SSDs) offer latencies in the sub-100μs range at reasonable cost [16, 14]. At the same time, improvements in network software and hardware have brought network latencies closer to their physical limits, enabling sub-100μs communication latencies. The net result of these developments is that the datacenter will soon be dominated by microsecond-scale I/O requests.

Device	Read	Write
Millisecond Scale		
10G Intercontinental RPC	100 ms	100 ms
10G Intracontinental RPC	20 ms	20 ms
Hard Disk	10 ms	10 ms
10G Interregional RPC	1 ms	1 ms
Microsecond Scale		
10G Intraregional RPC	300 μs	300 μs
SATA NAND SSD	200 μs	50 μs
PCIe/NVMe NAND SSD	60 μs	15 μs
10Ge Inter-Datacenter RPC	10 μs	10 μs
40Ge Inter-Datacenter RPC	5 μs	5 μs
PCM SSD	5 μs	5 μs
Nanosecond Scale		
40 Gb Intra-Rack RPC	100 ns	100 ns
DRAM	10 ns	10 ns
STT-RAM	<10 ns	<10 ns

Table 1: **I/O device latencies.** Typical random read and write latencies for a variety of I/O devices. The majority of I/Os in the datacenter will be in the microsecond range.

Today, an operating system uses one of two options when an application makes an I/O request: either it can block and poll for the I/O to complete, or it can complete the I/O asynchronously by placing the request in a queue and yielding the processor to another thread or application until the I/O completes. Polling is an effective strategy for devices with submicrosecond latency [2, 20], while programmers have used yielding and asynchronous I/O completion for decades on devices with millisecond latencies, such as disk. Neither of these strategies, however, is a perfect fit for microsecond-scale I/O requests: blocking will prevent the processor from doing work for tens of microseconds, while yielding may

reduce performance by increasing the overhead of each I/O operation.

A third option exists as a solution for dispatching I/O requests, *speculation*. Under the speculation strategy, the operating system completes I/O operations speculatively, returning control to the application without yielding. The operating system monitors the application: in the case of a write operation, the operating system blocks the application if it makes a side-effect, and in the case of a read operation, the operating system blocks the application if it attempts to use data that the OS has not read yet. In addition, the operating system may have a mechanism to rollback if the I/O operation does not complete successfully. By speculating, an application can continue to do useful work even if the I/O has not completed. In the context of microsecond-scale I/O, speculation can be extremely valuable since, as we discuss in the next section, there is often enough work available to hide microsecond latencies. We expect that storage class memories, such as phase-change memory (PCM), will especially benefit from speculation since their access latencies are unpredictable and variable [12].

Any performance benefit to be gained from speculation is dependent upon the performance overhead of speculating. Previous work in I/O speculation [9, 10] has relied on checkpointing to enable rollback in case of write failure. Even lightweight checkpointing, which utilizes copy-on-write techniques, has a significant overhead which can exceed the access latency of microsecond devices.

In this paper, we survey speculation in the context of microsecond-scale I/O devices, and attempt to quantify the performance gains that speculation has to offer. We then explore several techniques for speculation, which includes exploring existing software-based checkpointing techniques. We also propose new techniques which exploit the semantics of the traditional I/O interface. We find that while speculation could allow us to maximize the performance of microsecond scale devices, current techniques for speculation cannot deliver the performance which microsecond scale devices require.

2 Background

Past research has shown that current systems have built-in the assumption that I/O is dominated by millisecond scale requests [17, 2]. These assumptions have impacted the core design of the applications and operating systems we use today, and may not be valid in a world where I/O is an order of magnitude faster. In this Section, we discuss the two major strategies for handling I/O and show that they do not adequately address the needs of microsecond-scale devices, and we give an overview of I/O speculation.

2.1 Interfaces versus Strategies

When an application issues a request for an I/O, it uses an *interface* to make that request. A common example is the POSIX `write/read` interface, where applications make I/O requests by issuing blocking calls. Another example is the POSIX asynchronous I/O interface, in which applications enqueue requests to complete asynchronously and retrieve the status of their completion at some later time.

Contrast interfaces with *strategies*, which refers to how the operating system actually handles I/O requests. For example, even though the `write` interface is blocking, the operating system may choose to handle the I/O asynchronously, yielding the processor to some other thread.

In this work, we primarily discuss operating system strategies for handling I/O requests, and assume that application developers are free to choose interfaces.

2.2 Asynchronous I/O - Yielding

Yielding, or the asynchronous I/O strategy, follows the traditional pattern for handling I/O requests within the operating system: when a userspace thread issues an I/O request, the I/O subsystem issues the request and the scheduler places the thread in an I/O wait state. Once the I/O device completes the request, it informs the operating system, usually by means of a hardware interrupt, and the operating system then places the thread into a ready state, which enables the thread to resume when it is rescheduled.

Yielding has the advantage of allowing other tasks to utilize the CPU while the I/O is being processed. However, yielding introduces significant overhead, which is particularly relevant for fast I/O devices [2, 20]. For example, yielding introduces contexts switches, cache and TLB pollution as well as interrupt overhead that may exceed the cost of doing the I/O itself. These overheads are typically in the microsecond range, which makes the cost of yielding minimal when dealing with millisecond latencies as with disks and slow WANs, but high when dealing with nanosecond devices, such as fast NVMs.

2.3 Synchronous I/O - Blocking

Blocking, or the synchronous I/O strategy, is a solution for dealing with devices like fast NVMs. Instead of yielding the CPU in order for I/O to complete, blocking prevents unnecessary context switches by having the application poll for I/O completions, keeping the entire context of execution within the executing thread. Typically, the application stays in a spin-wait loop until the I/O completes, and resumes execution once the device flags the I/O as complete.

Blocking prevents the CPU from incurring the cost of context switches, cache and TLB pollution as well as interrupt overhead that the yielding strategy incurs. However, the CPU is stalled for the amount of time the I/O takes to complete. If the I/O is fast, then this strategy is optimal since the amount of time spent waiting is much shorter than the amount of CPU time lost due to software overheads. However, if the I/O is in the milliseconds range, this strategy wastes many CPU cycles in the spin-wait loop.

2.4 Addressing Microsecond-Scale Devices

Microsecond-scale devices do not fit perfectly into either strategy: blocking may cause the processor to block for a significant amount of time, preventing useful work from being done, and yielding may introduce overheads that may not have been significant with millisecond-scale devices, but may exceed the time to access a microsecond scale device. Current literature [2, 20] typically recommends that devices with microsecond ($\geq 5\mu$s) latencies use the yielding strategy.

2.5 I/O Speculation

Speculation is a widely employed technique in which a execution occurs before it is known whether it is needed or correct. Most modern processors use speculation: for example, branch predictors resolve branches before the branch path has been calculated [15]. Optimistic concurrency control in database systems enables multiple transactions to proceed before conflicts are resolved [5]. Prefetching systems attempt to make data available before it is known to be needed [8]. In all these speculative systems, speculation has no effect on correctness – if a misspeculation occurs either it has no effect on correctness or the system can rollback state as if no speculation had occurred in the first place.

I/O requests are a good speculation candidate for several reasons. The results of an I/O request are simple and predictable. In the case of a write, the write either succeeds or fails. For a read, the request usually returns success or failure immediately, and a buffer with the requested data is filled. In the common case, I/Os typically succeed – failures such as a disk error or an unreachable host are usually exceptional conditions that do not occur in a typical application run.

We depict the basic process of speculation in Figure 1. In order to speculate, a speculative context is created first. Creating a speculative context incurs a performance penalty ($t_{speculate}$), but once the context is created, the task can speculatively execute for some time ($t_{spec_execute}$), doing useful work until it is no longer safe to speculate (t_{wait}). In the meantime, the kernel can dispatch the I/O request asynchronously (t_{io_start}). Once the I/O request completes (t_{io_comp}), the kernel commits the speculative execution if the request is successful, or aborts the speculative execution if it is not.

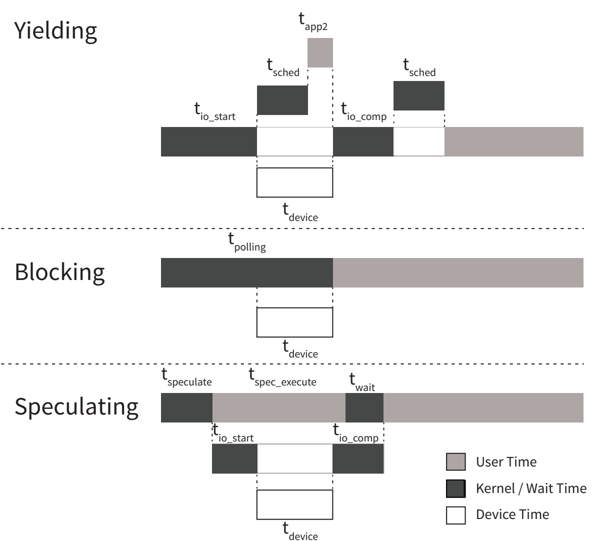

Figure 1: **Cost breakdown by strategy.** These diagrams show the relative costs for the yielding, blocking and speculating strategies.

In contrast to the blocking strategy, where the application cannot do useful work while the kernel is polling for the device to complete, speculation allows the application to perform useful work while the I/O is being dispatched. Compared to the yielding strategy, speculation avoids the overhead incurred by context switches.

This breakdown indicates that the performance benefits from speculation hinges upon the time to create a speculative context and the amount of the system can safely speculate. If the cost is zero, and the device time (t_{device}) is short, then it is almost always better to speculate because the CPU can do useful work while the I/O is in progress, instead of spinning or paying the overhead of context switches. However, when t_{device} is long compared to the time the system can safely speculate ($t_{spec_execute}$), then yielding will perform better, since it can at least allow another application to do useful work where the speculation strategy would have to block. When the time to create a context ($t_{speculate}$) is high compared to t_{device}, then the blocking strategy would be better since it does not waste cycles creating a speculative context which will be committed before any work is done.

For millisecond devices, yielding is optimal because t_{device} is long, so the costs of scheduling and context switches are minimal compared to the time it takes to dispatch the I/O. For nanosecond devices, blocking is optimal since t_{device} is short, so overhead incurred by either speculation or yielding will be wasteful. For microsecond devices, we believe speculation could be optimal if

Application	Description
bzip2	bzip2 on the Linux kernel source.
dc	NPB Arithmetic data cube.
dd	The Unix dd utility.
git clone	Clone of the Linux git repository.
make	Build of the Linux 3.11.1 kernel.
mongoDB	A 50% read, 50% write workload.
OLTP	An OLTP benchmark using MySQL.
postmark	E-mail server simulation benchmark.
tar	Tarball of the Linux kernel.
TPCC-Uva	TPC-C running on postgresql.

Table 3: **Applications.** A wide array of applications which we analyzed for speculation potential.

there are microseconds of work to speculate across, and the cost of speculating is low.

3 The Potential for Speculation

In order for speculation to be worthwhile, $t_{spec_execute}$ must be significantly large compared to the cost of speculation and the device time. In order to measure this potential, we instrumented applications with Pin [13], a dynamic binary instrumentation tool, to measure the number of instructions between I/O requests and the point speculation must block. For writes, we measured the number of instructions between a write system call and side effects-causing system calls (for example, `kill(2)` but not `getpid(2)`), as well as writes to shared memory. For reads, we measure the number of instructions between a read system call and the actual use of the resulting buffer (for example, as a result of a read instruction to the buffer), or other system call, as with a write. Our estimate of the opportunity for speculation is an extremely conservative one: we expect that we will have to block on a large number of system calls that many systems we discuss in section 4 speculate through. However, by limiting the scope of speculation, our findings reflect targets for speculation that produce a minimal amount of speculative state.

We instrumented a wide variety of applications (Table 3), and summarize the results in Table 2. In general, we found applications fell into one of three categories: pure I/O applications, I/O intensive applications, and compute intensive applications. We briefly discuss each class below:

Pure I/O applications such as `dd` and `postmark` performed very little work between side-effects. For example, `dd` performs a read on the input file to a buffer, followed by write to the output file repeatedly. On average, these applications perform on the order of 100 instructions between I/O requests and side effects.

We also looked at database applications including `TPCC-Uva`, `MongoDB` and `OLTP`. These applications are I/O intensive, but perform a significant amount of compute between side effects. On average, we found that these applications perform on the order of 10,000 instructions between read and write requests. These workloads provide an ample instruction load for microsecond devices to speculate through.

Compute intensive applications such as `bzip2` and `dc` performed hundreds of thousands to millions of instructions between side-effects. However, these applications made I/O calls less often than other application types, potentially minimizing the benefit to be had from speculation.

Many of the applications we tested used the buffer following a `read` system call immediately: most applications waited less than 100 instructions before using a buffer that was read. For many applications, this was due to buffering inside libc, and for many other applications, internal buffering (especially for the database workloads, which often employ their own buffer cache) may have been a factor.

4 Speculation Techniques

In the next section, we examine several techniques for speculation in the context of the microsecond era. We review past work and propose new design directions in the context of microsecond scale I/O.

4.1 Asynchronous I/O Interfaces

While asynchronous I/O interfaces [1] are not strictly a speculation technique, we mention asynchronous I/O since it provides similar speedups as speculation. Indeed, just as in speculation, program execution will continue without waiting for the I/O to complete. However, asynchronous I/O requires the programmer to explicitly use and reason about asynchrony, which increases program complexity. In practice, while modern Linux kernels support asynchronous I/O, applications use synchronous I/O unless they require high performance.

4.2 Software Checkpoint Speculation

In order to perform speculation, software checkpointing techniques generate a *checkpoint*, which is a copy of an application's state. To generate a checkpoint, we call `clone(2)`, which creates a copy-on-write clone of the calling process. After the checkpoint has been created, the system may allow an application to speculatively continue through a synchronous I/O call before it completes (by returning control to the application as if the I/O had completed successfully). The system then monitors the application and stops it if it performs an action which produces an external output (for example, writing

	Writes			Reads		
Application	Instructions	Calls/s	Stop Reason	Instructions	Calls/s	Stop Reason
Pure I/O Applications						
`postmark`	74 ± 107	518	`close`	15 ± 11	123	buffer
`make (ld)`	115 ± 6	55	`lseek`	$8{,}790 \pm 73{,}087$	180	`lseek` (31%) buffer (68%)
`dd`	161 ± 552	697	`write`	69 ± 20	698	`write`
`tar`	$248 \pm 1{,}090$	1,001	`write` (90%) `close` (9%)	144 ± 11	1,141	`write`
`git clone`	$1{,}940 \pm 11{,}033$	2,833	`write` (73%) `close` (26%)	14 ± 10	1,820	buffer
I/O Intensive Applications						
`MongoDB`	$10{,}112 \pm 662{,}117$	13,155	`pwrite` (94%)	62 ± 196	<1	buffer
`TPCC-Uva`	$11{,}390 \pm 256{,}018$	115	`write` (49%) `sendto` (22%)	37 ± 8	22	buffer
`OLTP`	$22{,}641 \pm 342{,}110$	141	`pwrite` (79%) `sendto` (7%)	31 ± 21	19	buffer
Compute Intensive Applications						
`dc`	$1{,}216{,}281 \pm 13{,}604{,}751$	225	`write`	$8{,}677 \pm 66{,}273$	156	buffer
`make (cc1)`	$1{,}649{,}322 \pm 819{,}258$	12	`write`	165 ± 21	431	buffer
`bzip2`	$43{,}492{,}452 \pm 155{,}858{,}431$	7	`write`	$1{,}472 \pm 345{,}827$	18	buffer

Table 2: **Speculation Potential.** Potential for speculation in the read/write path of profiled applications. We list only the stop reasons that occur in >5% of all calls. Error numbers are standard deviations, and "buffer" indicates that speculation was stopped due to a read from the read buffer.

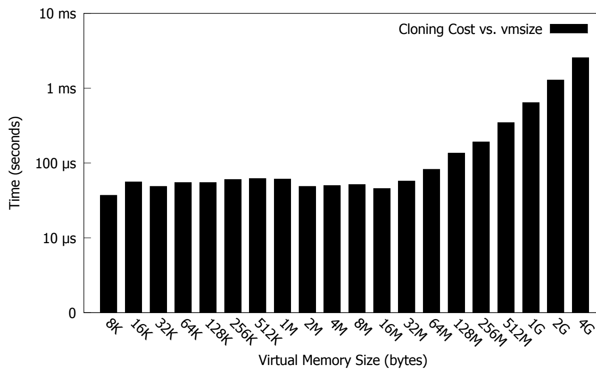

Figure 2: **Cloning Cost.** The cost of copy-on-write cloning for applications of various virtual memory sizes. Note that the axes are in log scale.

a message to a screen or sending a network packet) and waits for the speculated I/O to complete. If the I/O fails, the system uses the checkpoint created earlier to restore the application state, which allows the application to continue as if the (mis)speculation never occurred in the first place.

Software-based checkpointing techniques are at the heart of a number of systems which employ speculation, such as Speculator [9] and Xsyncfs [10]. These systems enabled speculative execution in both disks and over distributed file systems. These systems are particularly attractive because they offer increased performance without sacrificing correctness. Checkpoint-based speculation techniques hide misspeculations from the programmer, enabling applications to run on these systems unmodified.

However, providing the illusion of synchrony using checkpoints has a cost. We examined the cost of the clone operation, which is used for checkpointing (Figure 2). We found that for small applications, the cost was about 50μs, but this cost increased significantly as the virtual memory (vm) size of the application grew. As the application approached a vm size of 1GB, the cloning cost approached 1ms. While these cloning latencies may have been a small price to pay for slower storage technologies, such as disk and wide area networks, the cost of cloning even the smallest application can quickly eclipse the latency of a microsecond era device. In order for checkpoint-based speculation to be effective, the cost of taking a checkpoint must be minimized.

4.3 Hardware Checkpoint Speculation

Since we found checkpointing to be an attractive technique for enabling speculation given its correctness properties, creating checkpoints via hardware appeared to be a reasonable approach to accelerating checkpointing. Intel's transactional memory instructions, introduced with the Haswell microarchitecture [21] seemed to be a good match. Hardware transactional memory support has the potential of significantly reducing the cost of speculation, since speculative execution is similar to transactions. We can wrap speculative contexts into transactions which are

committed only when the I/O succeeds. Checkpoints would then be automatically created and tracked by hardware, which buffers away modifications until they are ready to be committed.

We examined the performance of TSX and found that the cost of entering a transactional section is very low (<20 ns). Recent work [18, 21] suggests that TSX transaction working sets can write up to 16KB and <1 ms with low abort rates (<10%). While TSX shows much promise in enabling fast, hardware-assisted checkpointing, many operations including some I/O operations, cause a TSX transaction to abort. If an abort happens for any reason, all the work must be repeated again, significantly hampering performance. While hardware checkpoint speculation is promising, finer-grained software control is necessary. For example, allowing software to control which conditions cause an abort as well as what happens after an abort would enable speculation with TSX.

4.4 Checkpoint-Free Speculation

During our exploration of checkpoint-based speculation, we observed that the created checkpoints were rarely, if ever used. Checkpoints are only used to ensure correctness when a write fails. In a system with local I/O, a write failure is a rare event. Typically, such as in the case of a disk failure, there is little the application developer will do to recover from the failure other than reporting it to the user. Checkpoint-free speculation makes the observation that taking the performance overhead of checkpointing to protect against a rare event is inefficient. Instead of checkpointing, checkpoint-free speculation makes the assumption that every I/O will succeed, and that only external effects need to be prevented from appearing before the I/O completes. If a failure does occur, then the application is interrupted via a signal (instead of being rolled back) to do any final error handling before exiting.

Unfortunately, by deferring synchronous write I/Os to after a system call, the kernel must buffer the I/Os until they are written to disk. This increases memory pressure and requires an expensive memory copy for each I/O. We continue to believe that checkpoint-free speculation, if implemented together with kernel and user-space processes to allow omitting the memory copy, will result in a significant performance increase for microsecond-scale devices.

4.5 Prefetching

While the previous techniques are targeted towards speculating writes, prefetching is a technique for speculating across reads. In our characterization of speculation potential, we found that speculating across read calls would be ineffective because applications are likely to immediately use the results of that read. This result suggests that prefetching would be an excellent technique for microsecond devices since the latency of fetching data early is much lower with microsecond era devices, reducing the window of time that a prefetcher needs to account for. We note that the profitability of prefetching also decreases with latency – it is much more profitable to prefetch from a microsecond device that a nanosecond device.

Prefetching already exists in many storage systems. For example, the Linux buffer cache can prefetch data sequentially in a file. However, we believe that more aggressive forms of prefetching are worth revisiting for microsecond scale devices. For example, SpecHint and TIP [3, 11] used a combination of static and dynamic binary translation to speculatively generate I/O hints, which they extended into the operating system [4] to improve performance. Mowry [7] proposed a similar system which inserted I/O prefetching at compile time to hide I/O latency. Since microsecond devices expose orders of magnitude more operations per second than disk, these aggressive techniques will be much more lucrative in the microsecond era.

4.6 Parallelism

Other work on speculation focuses on using speculation to extract parallelism out of serial applications. For example, Wester [19] introduced a speculative system call API which exposes speculation to programmers, and Fast Track [6] implemented a runtime environment for speculation. This work will likely be very relevant since microsecond devices expose much more parallelism than disk.

5 Discussion

As we have seen, a variety of different techniques exist for speculating on storage I/O, however, in their current state, no technique yet completely fulfills the needs of microsecond scale I/O.

Our study suggests that future work is needed in two areas. First, more work is needed to design appropriate hardware for checkpointing solutions. Second, the opportunity for checkpoint-free speculation needs to be studied in depth for both compute intensive and I/O intensive database applications.

6 Conclusion

This paper argues for the use of speculation for microsecond-scale I/O. Microsecond-scale I/O will soon

dominate datacenter workloads, and current strategies are suboptimal for dealing with the I/O latencies that future devices will deliver. Speculation can serve to bridge that gap, providing a strategy that enables I/O intensive applications to perform useful work while waiting for I/O to complete. Our results show that the performance of microsecond-scale I/Os can greatly benefit from speculation, but our analysis of speculation techniques shows that the cost of speculation must be minimized in order to derive any benefit.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. DGE-1144086, as well as C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[2] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 387–400. ACM, 2012.

[3] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[4] K. Faser and F. Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *USENIX Annual Technical Conference, General Track*, pages 325–338, 2003.

[5] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, pages 35–46, 1991.

[6] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A software system for speculative program optimization. In *Proceedings of the 7th annual IEEE/ACM International Symposium*, pages 157–168, 2009.

[7] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 3–17, New York, NY, USA, 1996. ACM.

[8] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th Conference on Parallel Architectures*, pages 135–145, 2004.

[9] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 24(4):361–392, 2006.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):6, 2008.

[11] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM, 1995.

[12] M. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras. Preset: Improving performance of phase change memories by exploiting asymmetry in write times. In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391. IEEE, 2012.

[13] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education*, 2004.

[14] S.-H. Shin, D.-K. Shim, J.-Y. Jeong, O.-S. Kwon, S.-Y. Yoon, M.-H. Choi, T.-Y. Kim, H.-W. Park, H.-J. Yoon, Y.-S. Song, et al. A new 3-bit programming algorithm using SLC-to-TLC migration for 8MB/s high performance TLC NAND flash memory. In *VLSI Circuits (VLSIC), 2012 Symposium on*, pages 132–133. IEEE, 2012.

[15] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, 1981.

[16] H. Tanaka, M. Kido, K. Yahashi, M. Oomura, R. Katsumata, M. Kito, Y. Fukuzumi, M. Sato, Y. Nagata, Y. Matsuoka, et al. Bit cost scalable technology with punch and plug process for ultra high density flash memory. In *IEEE Symposium on VLSI Technology*, pages 14–15. IEEE, 2007.

[17] H. Volos. Revamping the system interface to storage-class memory, 2012. PhD Thesis. University of Wisconsin at Madison.

[18] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[19] B. Wester, P. M. Chen, and J. Flinn. Operating System Support for Application-specific Speculation. In *Proceedings of the Sixth European conference on Computer systems*, pages 229–242. ACM, 2011.

[20] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel(R) Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2013.