Operating System I/O Speculation: How two invocations are faster than one

Keir Fraser, *University of Cambridge Computer Laboratory*[*]
Fay Chang, *Google Inc.*[*]
Keir.Fraser@cl.cam.ac.uk Fay@google.com

Abstract

We present an in-kernel disk prefetcher which uses speculative execution to determine what data an application is likely to require in the near future. By placing our design within the operating system, we provide several benefits compared to the previous application-level design. Not only is our system easier to implement and deploy, but by handling page faults as well as traditional file-access methods we are able to apply speculative execution to swapping applications, which often spend the majority of their execution time fetching non-resident pages. We also present two new OS features that further improve the performance of speculative execution for applications that have large page tables and working sets. These are a fast method for synchronizing an errant speculative process with normal execution, and a modified form of copy-on-write which preserves application semantics without delaying normal execution. Finally, by leveraging OS knowledge about memory usage and contention, we design a mechanism for estimating and limiting the memory overhead of speculative executions.

Our implementation for Linux 2.4.8 provides benefits of up to 60% for a wide range of explicit-I/O and swapping applications. Our results show that our support mechanisms for swapping applications provide significant performance benefits, and in some cases prevent speculative execution from hurting performance. We further demonstrate that our memory control mechanism effectively limits speculative overheads while allowing beneficial speculative executions to proceed unhindered.

1 Introduction

In the past decade, the gap between processing speeds and disk access times has increased by an order of magnitude [1]. At the same time, although memory sizes have increased substantially, so have application data requirements. Systems therefore continue to swap their data sets to and from disk as they are often too large to all fit in memory.

In recognition of this problem, there has been a great deal of research into automating disk prefetching algorithms that are dramatically more accurate than the standard heuristics in current operating systems. Recent work by Demke and Mowry [2] demonstrated impressive performance results using compiler-based techniques. However, system-wide use of their approach would require recompiling every application. Moreover, their compiler analyses are limited to looping array codes.

To address this problem, Chang and Gibson [3] proposed to discover the future data accesses of an application stalled on I/O by executing ahead in its code, ignoring non-memory-resident data. They also proposed a design for automatically modifying application binaries to apply this *speculative execution approach*. With this design, they demonstrated that speculative execution can deliver substantial performance benefits for a diverse set of applications that issue explicit file read calls.

Unfortunately, their user-level design has several major disadvantages. First, their design relies on the correct implementation of a complex binary modification tool. Second, while their design does not require application source code, applications must still be transformed to receive benefit. Third, their design can only issue prefetches for disk reads caused by explicit file I/O because page faults cannot be trapped by the application. The system therefore cannot be applied to *swapping* (or *out-of-core*) applications which leverage file- and swap-backed virtual memory to avoid the complexity of explicit I/O. Finally, their design does not measure or limit the memory used by speculative execution, and its resulting effect on system performance. It is therefore poorly suited for use on realistic systems, which do not always have abundant memory.

In this paper, we demonstrate how these problems can be overcome with an *in-kernel* design for automating speculative execution. We present a design that,

[*]This work was performed while the authors were at Compaq Systems Research Center.

by leveraging existing operating system features, is not only substantially easier to implement but also may provide benefit to arbitrary unmodified application binaries. Moreover, by exploiting knowledge that is typically unavailable outside the operating system, our design automates disk prefetching for virtual memory accesses as well as explicit I/O calls, enabling it to provide benefit regardless of the I/O-access methods used by applications. In addition, we propose two new operating system features which substantially improve the performance of speculative execution for swapping applications. Finally, we describe a mechanism for estimating the impact of memory use by speculative executions on system performance, and thereby controlling speculative execution when memory resources are not abundant.

We have implemented our design within the Linux 2.4.8 kernel and evaluated it using eight applications, which include four explicit-I/O applications, three swapping applications, and one application that performs a substantial amount of both forms of I/O. We demonstrate that our basic design provides similar benefit to the prior design on explicit-I/O applications, while requiring much less implementation effort. We then demonstrate that our design also provides benefit for swapping applications, particularly with the assistance of the new features we identified. Finally, by varying the amount of usable memory on our system, we demonstrate the benefit of our mechanism for controlling the memory usage of speculative execution.

The remainder of this paper is organized as follows. Section 2 describes the speculative execution approach to automating I/O prefetching. Section 3 describes the testbed and benchmarks that we use to evaluate our proposals throughout this paper. Section 4 presents and evaluates the baseline version of our new in-kernel design and implementation. Then, in Sections 5 and 6, we proceed to describe our improvements for swapping applications, and our mechanism for controlling memory overhead. Section 7 contrasts our in-kernel design with the prior user-level design. Finally, Sections 8 and 9 discuss related work, future work, and conclusions.

2 Speculative execution

The *speculative execution approach* exploits the increasing abundance of spare processing cycles to automate prefetching for applications that stall on disk I/O. Usually, when an application needs some data that is not in memory, it will issue a disk request and then stall waiting for that request to complete. Rather than simply wasting unused processing cycles while applications are stalled on I/O, the speculative execution approach uses these cycles to try to discover and initiate prefetching

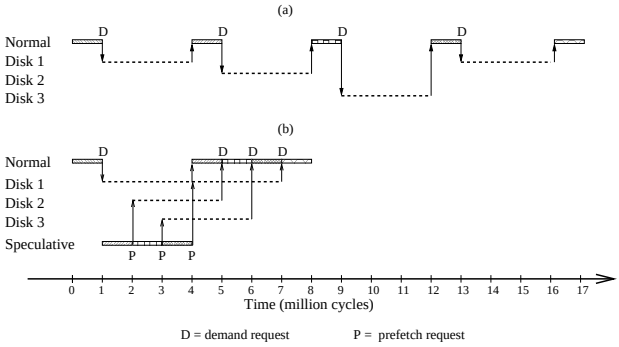

Fig. 1: Example illustrating how the speculative execution approach could reduce I/O stall time. (A) shows how execution would ordinarily proceed for a hypothetical application. (B) shows how execution might proceed for the application with the speculative execution approach. While normal execution is stalled on its first I/O request, speculative execution may be able to initiate prefetching for all the non-resident data that the application will access in the future. This could halve the application's execution time.

for the future data needs of stalled applications by running ahead of their stalled executions. In particular, the approach assumes that this speculative pre-execution of the application's code will be sufficiently similar to the application's future normal (non-speculative) execution that it will encounter the same accesses to non-resident data. Based on this assumption, speculative execution attempts to improve the application's subsequent performance by converting any such accesses to prefetches.

Figure 1 illustrates how this approach could deliver substantial performance improvements for a hypothetical application that accesses four non-resident data pages spread across three disks. For simplicity, assume that the application executes for one million cycles between each such access, and that a disk can service a request in three million cycles. When this application is executed, its execution will ordinarily alternate between processing and stalling on I/O. If the speculative execution approach were applied then, when normal execution stalls on its first I/O request, execution would continue speculatively. Whenever speculative execution encounters an access to non-resident data, it will instead issue a non-blocking prefetch call. In this manner, it may be able to initiate prefetching for all of the application's subsequent data accesses. When the original disk request completes, normal execution will resume. Now, however, its subsequent data accesses will be serviced out of main memory, halving the application's execution time.

It is worth noting that speculative execution will not be effective in all cases. For example, it will offer no benefit on systems where CPU, memory or disk are al-

ready fully utilized. Also, a speculative process will incorrectly predict future accesses if they depend on non-resident data. However, our success in applying speculative execution to a wide range of benchmark applications indicates that independent I/O accesses are common.

3 Experimental setup

Throughout this paper, we evaluate each successive design proposal after describing that proposal in order to isolate its performance impact and motivate further refinements. We present our experimental setup in this section to assist this progressive unfolding of our design.

We evaluate our design proposals on an 866MHz Pentium III configured to use 64MB of memory, running our modified Linux kernel. Most modern desktop computers have much more memory. We restricted the memory size deliberately to facilitate comparison with prior work [4, 3] and compensate for established I/O benchmarks [5] which use data sets that have not been updated to reflect growth in data set sizes. Our storage system consists of seven Compaq RZ1CB Ultra SCSI disks (12ms average access time). The file system is striped across four of the disks with a 64KB stripe unit. The central cylinders of the other three disks are designated as swap space. The maximum transfer rate supported by the disks and the SCSI interface is 40MB/s.

All our results are averages over three runs; however, the variance in execution time across these runs was always small (within a few percent of the calculated mean). All file- and swap-backed pages were flushed from memory before each run.

3.1 Benchmark applications

We use eight benchmark applications in our evaluation. Four of the benchmark applications are explicit-I/O applications, three are swapping applications, and one performs a substantial amount of both types of I/O. To assist comparison with prior work, these benchmarks are similar, and in many cases identical, to those used in prior evaluations of the TIP prefetching and caching manager [5], user-level speculative execution [3] and compiler-based prefetching [4, 2]. The benchmarks are summarized in Table 1 and described in greater detail below.

Agrep (version 2.0.4) is a fast pattern matching utility. Agrep opens each file on its command line in turn, and reads each one sequentially from start to finish. In the benchmark, Agrep searches 8971 files in the Linux 2.4.5 source tree for exact matches of a simple string that does not occur in any of the files.

Gnuld (version 2.11.2) is the Free Software Foundation's object code linker. Gnuld first reads each object file's file header and uses it to find the symbol header, which in turn provides offsets to the symbol and string tables. Gnuld then makes a small number of small, non-sequential reads to gather debugging information; the required file offsets are determined from the symbol tables. Finally, Gnuld sequentially reads the non-debugging sections in each object file. In the benchmark, an Alpha cross-linker is used to link 562 object files to produce a Digital UNIX kernel.

PostgreSQL (version 7.0.3) is an enhanced version of the original POSTGRES database management system. Our tests use a subset of the open-source database benchmark (OSDB), which implements the industry-standard AS3AP benchmark suite [6]. Our data set consists of four relations conforming to the AS3AP specification, which reside in 500MB of disk space. Our benchmark generates a set of indexes for each relation.

XDataSlice (version 2.2) is a data visualization package that allows users to view a false-color representation of arbitrary slices through a three-dimensional data set. The original application limited itself to data sets that fit into memory, but our version was modified for the TIP benchmark suite [5] to load data dynamically from large data sets. In the benchmark, XDataSlice retrieves 25 random slices through a set of 512^3 32-bit values which resides in 512MB of disk space.

FFTPDE and *MGRID* are two applications from the NAS Parallel benchmark suite [7], which have been modified by Demke and Mowry [2] so that their data sets do not fit in main memory. These applications make looping array accesses whose stride length and bounds vary dynamically during execution. This lack of predictability makes it hard for a conventional prefetcher to prevent I/O stalls.

MATVEC is a matrix-vector multiplication kernel that we obtained directly from Demke and Mowry's recent paper on compiler-based prefetching and memory management [2].

Finally, *Sphinx* is a speech recognition application. As with XDataSlice, the original application was modified to load its data dynamically from disk for the TIP benchmark suite [5]. The benchmark is to recognize an 18-second recording that was commonly used in Sphinx regression testing. The benchmark contains two phases: a first phase in which it reads about 15MB in a mostly sequential fashion from four data files, and a second phase of seemingly random accesses to a 176MB file.

Benchmark	Description	Run time	Read calls	Data set size
Agrep	text search	42 s	9385	105 MB
Gnuld	object code linker	30 s	16261	70 MB
PostgreSQL	AS3AP benchmark (database queries)	9915 s	4845047	535 MB
XDataSlice	visualization of 3-D data sets	171 s	46421	512 MB
FFTPDE	FFT solver for 3-D PDEs	5537 s	–	224 MB
MGRID	multigrid solver for 3-D potential	913 s	–	431 MB
MATVEC	Matrix-vector multiplication	1006 s	–	385 MB
Sphinx	Off-line speech recognition	72 s	66358	181 MB

Table 1: Benchmark characteristics. Run time is on an unmodified kernel, with no speculative execution.

4 In-kernel speculative execution

In this section we present and evaluate our basic design for leveraging speculative execution within the operating system. Our design is similar in spirit to the previous user-level design, but has the advantages of being much easier to implement and more accessible to users. Moreover, we demonstrate that not only does this design provide large benefits similar to those of the prior system for explicit-I/O applications, but also it can provide substantial benefits for swapping applications.

4.1 Basic design

We add a new type of process to the system – a *speculative* process. A speculative process is created by forking a normal process the first time it blocks on a disk request. A speculative process is completely destroyed only when its parent process exits. The operating system treats speculative processes differently from normal processes only in order to: 1) ensure that speculative execution is *safe* (that is, speculative processes do not produce output or otherwise change the results of executing applications); 2) enable speculative processes to issue prefetches on behalf of their parent processes; and 3) restrict the resource utilization of speculative processes so that they cannot hurt the performance of normal processes.

Notice that speculative processes are never created for normal processes that perform no disk I/O. We also allow users to opt out of speculative execution by setting a specific environment variable.

4.1.1 Safety

It is easy to ensure that speculative execution is safe because operating systems already severely restrict the ways in which different processes can affect one another. As a result, a system needs to restrict speculative processes in only three simple ways to ensure safety.

First, on most systems, a forked process shares file pointers with its parent process, and inherits write access to mapped files and shared memory segments. To ensure safety, when forking speculative processes, file pointers are instead copied, and write access to mapped files and shared memory segments is changed to copy-on-write access.

Second, systems typically establish a relationship between a parent and child process such that information about the child process is propagated to its parent. For example, when a child process is destroyed, the operating system typically delivers a signal to its parent. To ensure safety, we sever these ordinary relationships between speculative processes and their parents.

Finally, the system must restrict which system calls speculative processes can perform. For example, a speculative process must not be allowed to modify the file system or send signals to normal processes. Therefore, speculative processes are required to perform access checks before proceeding with system calls. If the requested system call is not safe (or *may* not be safe, since implementations should be conservative), the access checks either cause the speculative thread to return immediately, or alter the requested system call so that it is safe. For example, requests to map file regions with write access are converted to requests to map those file regions privately with copy-on-write semantics. It is easy to force speculative processes to return immediately from any unsafe system call as most operating systems contain a single access point for all system calls, such as a software trap handler. We modified this trap handler such that speculative processes perform a table lookup indexed by the system call number and then, depending on the value encoded in the table, either return immediately or continue with the system call. Altering system calls issued by speculative processes requires slightly more effort in the form of individually modifying the call-specific handler for each such call. However, this effort was required for only a small number of calls, such as `mmap` and "multiplexed" command interfaces such as `ioctl`.

4.1.2 Prefetching

A speculative process generates prefetch requests on behalf of its parent process by initiating a *non-blocking* prefetch whenever a normal process would have blocked on a disk read. Speculative execution then proceeds without the non-resident data. If the disk read resulted from an explicit file read call, any memory-resident data specified by the call is copied into the specified user buffer while regions of the buffer that would ordinarily be filled by non-resident data are unchanged. This allows the speculative process to make use of all available data during its subsequent execution. If the disk read resulted from a page fault then the system, as usual, allocates a page frame, updates the appropriate page table entry to map the new page frame, and initiates a disk read of the appropriate data. However, rather than blocking until the disk read completes, the speculative process returns from the fault immediately so that it can run ahead of its parent process. Since its page table has been updated, any subsequent accesses it makes to the same page will not generate another page fault.

It is possible that a speculative process will execute, but not succeed at generating accurate prefetches for its parent. This may occur because future execution depends on non-resident data, a system call that speculative processes are not allowed to perform, or inter-process communication through shared memory (since, as discussed in the prior section, shared pages are mapped copy-on-write in speculative processes). To increase the chances that the speculative process will generate useful prefetches, whenever the parent process is about to block waiting for some data for which its speculative child failed to generate a prefetch, it attempts to *synchronize* its speculative child, where synchronizing is just like forking except that we reuse the speculative child's process structure. A parent process can easily detect when its speculative child failed to generate a prefetch because a process must first allocate a page frame to hold data before issuing a disk read for that data. Therefore, if no page frame has been allocated for some data, then no prefetch has been issued for that data. We attempt to hide the observed cost of synchronization within the time to fetch the data from disk, during which the parent process would ordinarily block, by issuing the disk request before we begin synchronizing.

On a typical system, a process initiates file readahead while servicing a file read system call, and initiates page cluster reads when it faults on a page that is neither in memory nor already being fetched from disk. However, if a speculative process is issuing the correct prefetches, then the default prefetch heuristics are at best redundant and might waste memory and disk bandwidth by prefetching unneeded data. We therefore disable these heuristics if a speculative process has prefetched the data that its parent process is requesting.

4.1.3 Basic resource control

Supporting speculative execution consumes processing cycles, disk bandwidth, and memory. Ideally, speculative execution only uses resources that would otherwise be wasted, so it cannot hurt system performance. This section describes how we restrict the processing time of speculative processes, and the disk bandwidth and memory of speculative prefetches.

Processor cycles. We ensure that speculative execution does not steal processing cycles from normal processes by only scheduling speculative processes when nothing else is runnable. Furthermore, speculative processes are preempted as soon as a normal process becomes runnable. Rather than relying on the operating system's existing priority mechanism, which does not guarantee the desired behavior, we implement this with a simple modification (six additional lines) in the kernel scheduler code.

Prefetching resources. A speculative prefetch can steal disk bandwidth from normal processes by delaying a disk request from a normal process. Morever, premature prefetching can hurt performance, even if all the prefetches are accurate, by causing useful data to be unnecessarily ejected from memory. As suggested by Patterson, et. al. [5], we limit the maximum delay of a normal request by only issuing a prefetch to a disk which has no more than one request outstanding. We also use their idea of a *prefetch horizon* – a system-dependent, calculable maximum number of prefetches in advance beyond which there is unlikely to be a benefit to initiating a prefetch – to throttle speculative processes that are generating prefetches too quickly. If a speculative process attempts to prefetch further ahead than the prefetch horizon, it is marked non-runnable until its parent process either accesses some of the data it prefetched, or synchronizes it.

Finally, to avoid wasting resources, if a speculative process begins executing process termination code (e.g. as a result of issuing an `exit` system call or generating a terminating exception), then the operating system checks whether its parent process is terminating. If not, then the speculative process is not allowed to terminate; instead, its memory is reclaimed and it is marked non-runnable until its parent process synchronizes it.

4.2 Performance of the basic design

Figure 2 shows the overall performance of the basic design. For all four explicit-I/O applications, shown in the leftmost section, our in-kernel design delivers large

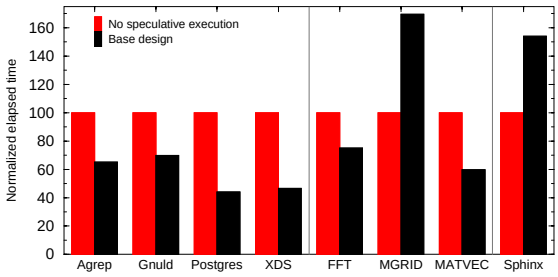

Fig. 2: Performance of our basic in-kernel design.

improvements, reducing elapsed times by 35% to 56%. These benefits are comparable to those delivered by the user-level design [3], and are achieved for the same reasons. In particular, unlike file readahead, the speculative execution approach enables prefetching across files and can leverage the decision paths encoded in applications to generate accurate prefetches for accesses that are seemingly random.

The results for swapping applications, shown in the central section, and our combination application, Sphinx, are more varied. We deliver substantial benefits for FFTPDE and MATVEC, but we degrade the performance of MGRID and Sphinx. Thus, our base design performs poorly compared with the compiler-based approach [4], which delivered a substantial performance benefit for MGRID.

Table 2 provides more detailed information about the executions. Unsurprisingly given the overall results, for all the applications except MGRID and Sphinx, speculative execution significantly reduces both the number of I/O stalls and the I/O stall time experienced by normal execution.

One potential concern with speculative execution is that it will not generate prefetches early enough to hide a substantial amount of I/O stall time. The figures for full speculative prefetches show that the vast majority of speculative prefetches actually complete before the data is accessed during normal execution; in other words, there are very few partial stalls on in-progress prefetches. Another potential concern is that, as with any heuristic approach, speculative execution may generate prefetches for data that will not be used, wasting both memory and disk bandwidth. The figures for unused speculative prefetches show that speculative execution is perfectly accurate for Agrep and MATVEC. For the other benchmarks, speculative execution generates some needless prefetches, but is always much more accurate than the operating system's default file readahead and page cluster heuristics. Furthermore, because good speculative prefetches disable the operating system's default prefetching heuristics, we are able to avoid a large proportion of needless prefetches for all benchmarks except Sphinx. This further helps performance by reducing contention for memory and disk bandwidth.

On the other hand, comparing the figures for explicit-I/O and swapping applications reveals that synchronization is substantially more expensive for swapping applications. In particular, comparing the synchronization times to original execution times (shown in the first column) reveals that MGRID is synchronizing for almost half of its original execution time. This suggests one way in which the base design is inefficient for swapping applications, and ineffective for MGRID in particular. For many applications, we also notice a substantial increase in the number of copy-on-write faults. Finally, Sphinx demonstrates a different potential problem with speculative execution. The memory use of speculative execution can cause useful data to be prematurely ejected from memory. This is revealed by how speculative execution *increases* the total number of I/O stalls for Sphinx. We address these weaknesses of the basic design in the next two sections.

5 Improved prefetching for swapping applications

Although the basic design discussed in the previous section works well for explicit-I/O applications, it can hurt the performance of applications that exploit virtual memory. In this section, we discuss two simple additions to standard operating system mechanisms which can greatly improve prefetching performance for memory-intensive applications.

5.1 Fast, preemptible refork

Since they rely on file- and swap-backed virtual memory to hide I/O, swapping applications typically have very large page tables. This presents two problems. First, when synchronizing its speculative child, a parent process can be substantially delayed by the time required to release its speculative child's old state and then make a fresh copy of the parent process's state. Second, the cycles consumed in synchonization reduce the number of spare processing cycles in which speculative execution can make progress. We reduce the cost of synchronization by adding a fast, preemptible refork operation.

Recall (from Section 4.1.2) that the parent process begins synchronizing its speculative child only after issuing a disk request that would ordinarily cause it to block. If the synchronization operation does not complete before the disk request completes, then the parent process will no longer need to block after synchronizing its child.

Benchmark	Specx enabled?	I/O Stalls		Spec prefetches		Unused prefetches		Sync delay	CoW faults
		Total	Time	Total	Full	Spec	Readahead		
Agrep	No	14080	30s	–	–	–	50	–	286
(42s)	Yes	4239	17s	7350	92%	0	50	0s	326
Gnuld	No	5434	26s	–	–	–	5245	–	8
(30s)	Yes	3228	15s	4684	74%	641	472	2s	8
PostgreSQL	No	1056013	8471s	–	–	–	823234	–	259
(9915s)	Yes	267498	3230s	1201033	87%	25101	43011	45s	2714
XDS	No	22887	159s	–	–	–	178430	–	13
(171s)	Yes	5905	63s	44512	87%	634	1632	7s	11376
FFTPDE	No	439221	5345s	–	–	–	3188757	–	8
(5537s)	Yes	201384	3024s	1384548	86%	583	116134	929s	391281
MGRID	No	87594	757s	–	–	–	130413	–	8
(913s)	Yes	59732	1020s	1095623	90%	14	21390	428s	179714
MATVEC	No	105518	925s	–	–	–	332361	–	7
(1006s)	Yes	36114	375s	893238	80%	0	12788	75s	91
Sphinx	No	6238	43s	–	–	–	19711	–	1386
(72s)	Yes	6957	58s	8417	68%	909	20831	6s	1388

Table 2: Effectiveness of prefetching by the basic design, compared with a system which performs no speculative execution. `I/O stalls` is the number of, and elapsed time during, I/O stalls experienced by normal execution. `Speculative prefetches` is the number of speculative prefetches, and the percentage of those prefetches that completed before the data was requested by normal execution. `Unused speculative prefetches` is the number of speculative prefetches that prefetched data that was not used before being ejected from memory. `Unused readahead prefetches` is the same statistic for prefetches generated by the operating system's file readahead and page clustering heuristics. `Sync delay` is the total time used to synchronize the speculative child process, some of which is hidden by disk access latency. `CoW faults` is the total number of copy-on-write faults experienced during normal execution.

Since the speculative process is not allowed to steal cycles from non-speculative processes, this means that the speculative process will not be able to run ahead of the parent process. (At best, on a multiprocessor, it may run in tandem with its parent). Therefore, there is no benefit to requiring that the parent complete a synchronization, and we are better off ensuring that synchronization does not needlessly delay a parent process. We accomplish this by periodically checking whether the disk request has completed. If the read has completed, the parent simply stops its synchronization attempt, and the speculative child continues to be non-runnable. The parent will attempt to complete a synchronization the next time it is delayed by disk I/O.

While this usually hides the cost of synchronization from the parent, it may *increase* the synchronization delay perceived by the child. However, we observe that the parent process will attempt to synchronize its child every time it needs to access any data that is not in memory. Swapping applications, which typically have a large working set, will therefore synchronize quite often, and so the page tables are unlikely to have changed significantly. This allows us to reduce synchronization time by releasing and updating only those page table entries that have changed in the parent or the speculative child since the last synchronization. Moreover, this optimization complements preemptible reforking; if the parent cannot complete a synchronization while one of its disk requests is being serviced, the partial synchronization is likely to reduce the amount of work it must perform to complete the synchronization the next time it is delayed by disk I/O.

5.2 One-way copy-on-write

If synchronization follows the usual forking semantics, the parent process will experience a copy-on-write page fault for each page that it attempts to modify before its speculative child. Copy-on-write faults can introduce substantial delay because of the page allocation and data copy that are required. In particular, although the cost of page allocation is generally quite low, it can increase dramatically when memory contention is high [2]. Observing that safety only requires that a page be copied if the child attempts to modify it, we avoid adding page allocations to parent processes by adding support for one-way copy-on-write; that is, we allow normal processes to make modifications which may be observed by speculative process.

Supporting one-way copy-on-write requires only a few modifications. First, while synchronizing, only the speculative child's page table entries and memory region mappings are marked as copy-on-write. We also add a speculative reference count on page frames and

Benchmark	Refork type	Refork time		Refork attempts		
		Total	Mean	Total	Completed (%)	
FFTPDE	Normal	929 s	7 ms	136645	136645	(100%)
	Fast	274 s	2 ms	118581	105879	(89%)
MGRID	Normal	428 s	12 ms	37065	37065	(100%)
	Fast	144 s	3 ms	48668	39583	(81%)
MATVEC	Normal	178 s	10 ms	17669	17669	(100%)
	Fast	75 s	3 ms	24951	17083	(68%)
Sphinx	Normal	13 s	3 ms	3769	3769	(100%)
	Fast	6 s	2 ms	2465	2254	(91%)

Table 3: Synchronization cost: the effect of a fast, preemptible re-fork.

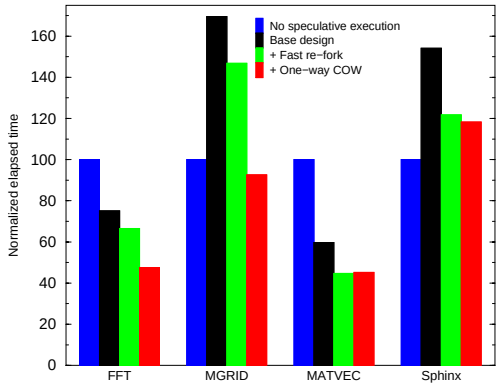

Fig. 3: Performance of speculative prefetching with fast re-forking and one-way copy-on-write.

swap slots, which track the number of references from speculative processes. When servicing a page fault during normal execution for a resident swap-backed page, we subtract the speculative reference count from the total count to determine whether the page must be copied. Finally, whenever a speculative process services a copy-on-write fault, we decrement the reference count on the original page.

5.3 Swapping application performance

Fast, preemptible reforking and one-way copy-on-write have negligible effect (less than one percent) on the execution times of our explicit-I/O applications, which use a modest amount of virtual memory. Figure 3 shows the degree by which our swapping applications benefit from these mechanisms. All four swapping applications run significantly faster compared with our baseline approach, with the speedup across applications approximately equally attributed to the two mechanisms. Moreover, while MGRID runs 60% slower with baseline speculative prefetching, fast, preemptible reforking and one-way copy-on-write eliminate this overhead.

Benchmark	CoW faults		
	No Specx	Basic	One-way
FFTPDE	8	391281	8
MGRID	8	179714	8
MATVEC	7	91	7
Sphinx	1386	1388	1387

Table 4: The effect of one-way copy-on-write on the number of copy-on-write faults experienced during normal execution. `Basic` is our base speculative execution design. `One-way` includes the one-way copy-on-write optimization. `No Specx` has all speculative execution disabled.

Detailed information about the benefits of fast, preemptible reforking are presented in Table 3. For the three scientific applications, the improvement over our baseline approach is dramatic: total refork time is reduced by a factor of three or more. Importantly, unlike normal reforks, the average fast refork time is much shorter than an average disk access for all of the benchmarks. Faster reforking enables a high proportion of these preemptible refork attempts to complete, and provides speculative execution with more time to run before it is preempted by normal execution.

Fast, preemptible reforking can increase the number of synchronization attempts (as with both MGRID and MATVEC) because preempted attempts will quickly be retried, the next time normal execution stalls. This mechanism increases the number of *completed* synchronizations, however, only for MGRID. In terms of execution time, this increase is far outweighed by the reduced refork time.

Examination of detailed application traces reveal that the improved performance of Sphinx is due to reforks being preemptible. Not only does this prevent normal execution from being needlessly delayed, but also it reduces the time during which the speculative process is runnable. Because Sphinx has a large memory footprint, leaving speculative execution non-runnable can substantially reduce memory contention, which is the main reason for degraded performance during this benchmark.

Table 4 shows that the one-way copy-on-write mechanism delivers dramatic reductions in the number of copy-on-write faults during normal execution for FFT-PDE and MGRID. The performance benefit of these reductions can be seen from the results in Figure 3. These performance improvements are due not only to the direct benefit of fewer copy-on-write faults, but also to the indirect benefit of decreasing memory contention by requiring fewer page allocations. MATVEC and Sphinx gain no noticeable benefit from one-way copy-on-write because, even with this mechanism disabled, normal execution experiences very few copy-on-write faults due to speculative execution. (Each benchmark experiences some unavoidable number of copy-on-write faults as a result of write accesses within shared libraries, which can be seen from the count of copy-on-write faults when speculative execution is disabled.)

6 Controlling memory overhead

The simple resource control mechanisms in the basic design (Section 4.1.3) are sufficient for a system that always has abundant memory. Most systems, however, are not so over-supplied. Further, it seems likely that the performance of memory-intensive applications would be harmed by ineffective speculative execution. In this section, we describe our mechanism for controlling the memory overhead incurred by speculative processes. This mechanism enables practical speculative execution on systems which may experience memory contention.

It is difficult to control memory overhead while still enabling effective speculative execution because, unlike processor or disk bandwidth, it is usually not possible to determine whether the resource is actually being wasted, and can therefore be used without hurting system performance. For example, even if the memory mapped by all extant processes is much less than the total amount of memory in the system, there is always a chance that a page that contains file data will be re-accessed.

The previous user-level design relied on the TIP prefetching and caching manager [5]. TIP performs *cost-benefit analysis* to determine when allocating some memory for prefetching would be more beneficial than allowing the LRU file cache to retain that memory. TIP is not a complete memory management solution for speculative execution, however, because speculative processes need to allocate memory not only to hold prefetched data, but also to dynamically allocate memory and make modifiable copies of pages to which they have copy-on-write access. Furthermore, the benefit of allowing a speculative process to allocate memory for its own use depends entirely on whether it will subsequently be able to issue useful prefetches, which depends on factors not within its control, such as how often it will be preempted.

We preserve the principle idea of a cost-benefit framework but, rather than building a complicated system model to predict the benefit of each speculative allocation request, we propose a simpler *reactive* approach to controlling memory overhead. Specifically, we estimate the benefit that a speculative process has already provided by prefetching data, and the cost it has already incurred by its memory consumption. This allows us to estimate the current benefit or cost of each speculative process and, by disabling processes accordingly, restrict the overhead incurred by ineffective speculative execution.

6.1 Reactive cost-benefit analysis

We estimate the cost and benefit of each speculative process with the assistance of an *eviction list* (or *ghost buffer* [8]). The eviction list tracks the non-resident pages that would most likely be in memory if no speculative execution had taken place. Each list entry uniquely identifies the disk block of one such page. To determine how many entries the list should contain, we track the number of *speculative pages* that are in memory only because of speculative execution. The eviction list is a FIFO to which we add an entry for each non-speculative page that is evicted from memory, and remove an entry if the FIFO is already full when a new page is added.

The eviction list allows the system to estimate the number of I/O stalls that speculative executions added to, or removed from, normal executions. In particular, when a normal process requests a disk read, if there is a matching entry in the eviction list, then we have identified an *added stall* that would not have occurred in the absence of speculative execution. Conversely, when a normal process avoids a disk read by using a speculative page, we have identified a *removed stall* that was prevented by speculative execution. The eviction list further aids accurate identification of removed stalls by allowing the system to detect when a speculative prefetch is merely refetching previously-resident data that was only evicted because of speculative memory use. Figure 4 describes the updates and accesses that can occur to the eviction list and associated performance counts in greater detail.

With the resulting per-process counts of removed stalls ($RemovedStalls_p$), and the system-wide count of added stalls ($TotAddedStalls$), we could estimate the net overhead of each speculative process as the overall cost/benefit for its execution so far. We could then use this measure to disable a speculative process whenever its net estimated overhead is above some selected

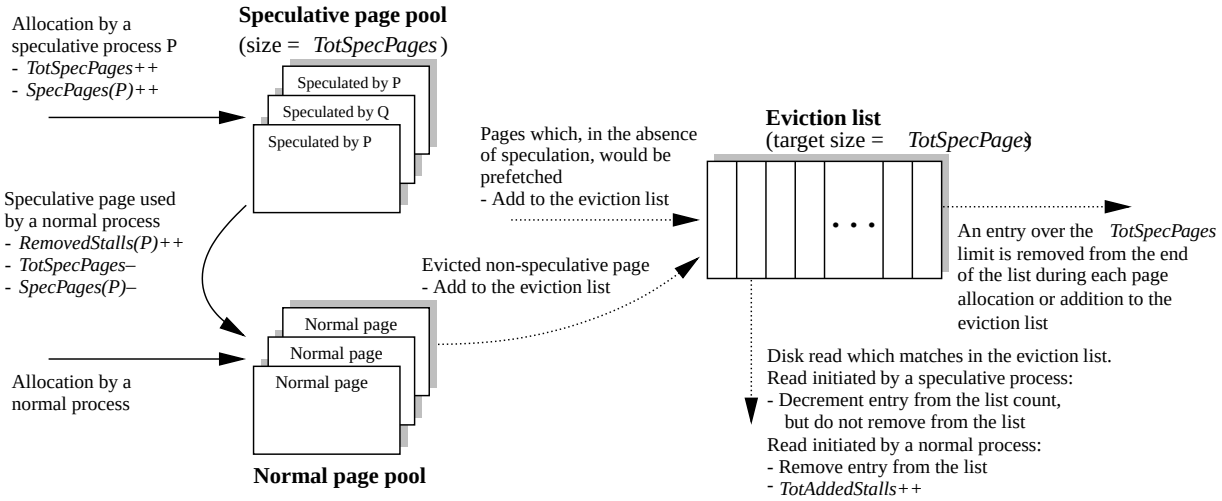

Fig. 4: Using the eviction list to calculate added and removed stalls. Many page allocations and evictions update or query the eviction list. This diagram enumerates the possible transitions that can occur. Dotted lines represent page identifiers being added to or removed from the eviction list.

threshhold. Unfortunately, this approach would be unresponsive to changes in the effectiveness of a speculative process. Such changes might be caused by different phases within the application which affect prefetching accuracy, or simply by varying memory contention from concurrent processes. We therefore use an approach that not only bounds the estimated overhead of enabled speculative processes, but also is responsive to changes in the estimated overhead of speculative processes.

The remainder of this section describes the *reactive* approach that we implemented and evaluated. Many other possibilities exist, but a survey is beyond the scope of this paper.

We divide system time into periodic *intervals*, where t denotes the current interval. At the end of each interval, a pair of cost-benefit estimates are updated for each speculative process based on the estimates and accumulated stall counts from the previous interval, and whether the speculative process was enabled or disabled during that interval. We use these estimates (and any accumulated stall counts since the last interval) to estimate the overhead at any time, in order to decide when to enable or disable each speculative process, as follows.

If a speculative process was *enabled* (i.e. marked runnable), we exponentially decay the previous stall estimates at the end of each interval:

$$\begin{aligned} Cost_p(t) &= (1-\alpha) \times Cost_p(t-1) + \\ &\quad \alpha \times AddedStalls_p(t-1) \\ Benefit_p(t) &= (1-\alpha) \times Benefit_p(t-1) + \\ &\quad \alpha \times RemovedStalls_p(t-1) \end{aligned}$$

Per-process counts of removed stalls are maintained directly by observing when speculative pages are first referenced by a normal process. However, to calculate per-process *added* stalls, the system-wide total for each interval is divided among the speculative processes in proportion to their memory use:

$$AddedStalls_p(t) = TotAddedStalls(t) \times \frac{SpecPages_p}{TotSpecPages}$$

We then combine the cost-benefit estimates in the simplest manner to obtain the *overhead* of each enabled speculative process:

$$Overhead_p(t) = Cost_p(t) - Benefit_p(t)$$

If a speculative process's estimated overhead is non-negligible, we mark it non-runnable and reclaim its memory. Therefore, if memory is tight, speculative processes that consume more memory will need to prefetch more effectively to remain runnable.

If a speculative process was *disabled* (i.e. marked non-runnable), we simply add to the previous cost and benefit estimates at the end of each interval:

$$\begin{aligned} Cost_p(t) &= Cost_p(t-1) + \\ &\quad AddedStalls_p(t-1) \\ Benefit_p(t) &= Benefit_p(t-1) + \\ &\quad RemovedStalls_p(t-1) \end{aligned}$$

Our overhead calculation then includes the period over which the process has been stopped. Let s be the interval during which non-runnable process p was last executed:

$$Overhead_p(t) = \frac{Cost_p(t) - Benefit_p(t)}{t-s}$$

Thus we decay the overhead estimate over time, which allows the system to set an upper bound on the net estimated overhead before a speculative process should be

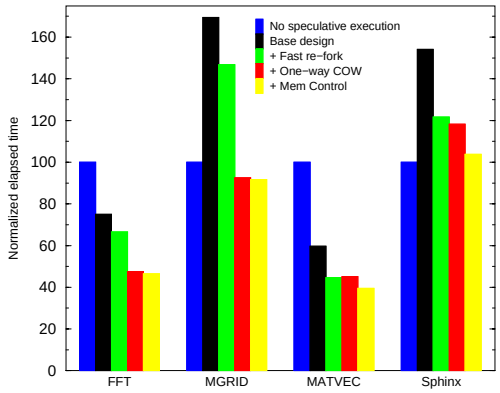

Fig. 5: Overall benefit of memory control for swapping applications (with the default machine configuration).

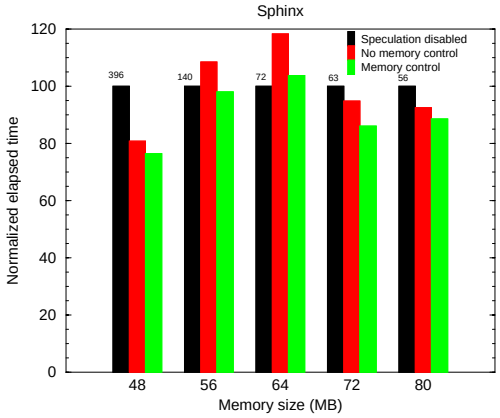

Fig. 6: The effect of memory control on the Sphinx benchmark, for a range of memory configurations. Run times for each memory size are normalized to the execution time with no speculation. These times are shown above the corresponding bar, in seconds.

allowed to run again. Notice that this property is independent of the manner in which the stall counts are decayed for enabled speculative processes.

There is some scope for investigating more informed techniques for reenabling speculative processes; for example, we could allow a speculative process to continue running and issuing prefetch requests, but prevent these requests from reaching the disk or allocating memory. This would reduce resource pressure compared with a fully-enabled speculative process, while providing us with more information to help determine when it would be worthwhile to reenable speculative execution. However, a major benefit of our current approach is its *conservativism* — a more informed approach would risk a harmful increase in resource pressure for just those applications that require most care.

Mem size	Spec	M	Threshold crossings	Syncs	Stall time
48 MB	Off	-	-	-	315s
	On	N	-	18303	224s
	On	Y	15	14790	218s
56 MB	Off	-	-	-	106s
	On	N	-	8356	102s
	On	Y	12	1963	101s
64 MB	Off	-	-	-	43s
	On	N	-	3337	47s
	On	Y	10	1979	41s
72 MB	Off	-	-	-	34s
	On	N	-	2282	25s
	On	Y	4	957	23s
80 MB	Off	-	-	-	28s
	On	N	-	1955	21s
	On	Y	5	729	20s

Table 5: The effect of memory control on synchronization attempts (`Syncs`) and I/O stall time. Three sets of results are shown for each memory configurations: speculation disabled, speculation enabled, and speculation with memory control (`M`) enabled. Threshold crossings is the number of times speculative execution is disabled due to intolerable memory overhead. Execution times without speculative execution are shown in Figure 6.

6.2 Evaluation of memory control

Figure 5 shows the performance benefit of memory control for our swapping applications. For our benchmarks, memory control only has a significant effect for Sphinx. This indicates that, as desired, memory control does not diminish the performance benefit of speculative execution when speculative execution is effective.

Figure 6 shows Sphinx results when the system is configured to have differing amounts of usable memory. In all memory configurations, our mechanism provides some benefit to Sphinx as compared to having no memory control mechanism. Moreover, in 56MB and 64MB configurations, memory control is able to eliminate, or significantly reduce, the performance penalty that occurs without memory control. These results suggest that this mechanism can effectively prevent speculative execution from harming performance in cases where speculative prefetching has too few resources to provide benefit. This is an important requirement if a prefetching system is to be deployed ubiquitously in a system.

From the detailed information listed in Table 5, it is evident that much of the performance benefit comes from reduced stall time. However, further gains are possible due to reduced memory contention in the absence of speculative execution. This will, for example, reduce the number of soft page faults experienced by normal execution for pages which are in the process of being

laundered, and can also reduce the cost of page allocations.

However, the adverse effects of speculative execution can linger for some time after it has been disabled, as the resulting 'gap' in memory cannot be immediately filled with useful data. Furthermore, in estimating overhead, our mechanism currently assumes average stall times. Actual stall times can vary greatly, which is why memory control does not entirely eliminate the penalty of speculative execution in our 64MB results.

Our mechanism disables speculative execution quite infrequently. However, the number of synchronization attempts is considerably reduced, indicating that, when speculation is disabled, it remains disabled for a considerable period of time. This is due to our conservative algorithm, which ensures that speculation is only reenabled when its net estimated overhead is negligible.

Surprisingly, speculative execution improves application performance on a 48MB system without the benefit of the memory control mechanism. This is due to more accurate prefetching compared with the default readahead heuristic; the memory cost of speculative execution is more than offset by the reduction in needlessly prefetched data. However, even in this case, memory control still provides a further gain of nearly 10% by disabling speculative execution while it is less effective. In the 72MB and 80MB results, basic speculative execution has enough memory available to provide overall benefit; however, the control mechanism is still able to identify a handful of places where it is beneficial to temporarily disable speculative execution.

7 Comparison to previous design

This paper describes an in-kernel design for applying the speculative execution approach to arbitrary, unmodified executables. In contrast, the previous design [3, 9], called *SpecHint*, specifies an automatable procedure for modifying application binaries to apply the speculative execution approach.

The SpecHint design has two advantages compared with an in-kernel design. First, SpecHint requires no operating system support specific to speculative execution, allowing deployment on systems where OS modifications are not feasible. Second, SpecHint can exploit static analyses and transformations to specialize the application code for speculative execution. For example, calls to expensive library functions, such as `printf`, can be removed to speed up speculative execution. More complex analysis might remove loops with data-dependent bounds which could trap speculative execution and prevent it from generating further I/O hints.

However, in addition to increasing the accessibility of speculative execution by providing it as an operating system service, an in-kernel design has four major advantages relative to the SpecHint design.

1. Unlike SpecHint, our design can be applied to applications that implicitly generate I/O via page faults to swap space or mapped files.

2. While SpecHint assumed that systems always have abundant spare memory, our design can estimate its effect on the memory performance of non-speculative executions.

3. As discussed in a previous report [9], SpecHint makes some assumptions in its attempts to ensure that the modified binary will not produce different results than the original binary. While these assumptions will hold in most cases, an in-kernel design can guarantee that adding speculative executions will not introduce errors in normal execution.

4. Binary modification tools are difficult to implement correctly and in a compiler-independent manner. Chang [9] reports that an implementation of SpecHint required 19,000 lines of C code and 6,000 lines of assembly code. That implementation transformed only statically-linked, single-process Alpha binaries produced using the native *cc* compiler for Digital Unix 3.2. In contrast, our implementation required only 5,000 lines of C code and 50 lines of assembly code, of which 90% is confined to two new files and memory management modifications, and it can be applied to arbitrary Linux x86 executables. Moreover, since our implementation contains fewer than 100 lines of x86-specific code, it should easily extend to handle arbitrary Linux executables on other platforms as well.

8 Related and future work

Common access pattern heuristics [10, 11, 12, 13] prefetch according to a small set of access patterns that occur frequently, such as sequential access. The system checks whether recent accesses fit a known pattern and, if so, issues prefetches by extrapolating the pattern. The simplicity of these approaches is an advantage. However, if accesses do not fit a known pattern then the heuristic cannot help, and may even hurt, application performance.

Dynamic history-based approaches [14, 15, 16, 17, 18, 19] initiate prefetching based on patterns inferred from previous access sequences. Although these systems can discover new patterns and exploit this knowledge across multiple applications, they cannot help with

non-repetitive accesses and may need to observe a large number of data accesses before an accurate pattern can be inferred. Furthermore, the history data can itself occupy a large amount of memory.

In *static analysis-based* approaches [20, 4], a compiler analyzes the application to deduce the accesses it will make during execution. It then inserts hints into the application to inform a run-time prefetcher. Although these approaches have low run-time overhead, the required interprocedural analysis is difficult. As a result, existing systems benefit only looping array codes. Furthermore, system-wide deployment will require all applications to be recompiled.

Cao *et al.* [21] note that aggressive prefetching strategies can do more harm than good by evicting data from memory which is later accessed. They propose a set of rules that an integrated prefetching and caching policy should follow to avoid hurting performance, and suggest two conforming policies. However, these policies assume perfect knowledge of application reference streams. Later work [22] mentions, but does not evaluate, possible heuristics for accomodating imperfect reference streams.

The TIP prefetching system [5] uses a cost-benefit model to control prefetching into, and eviction from, a file cache of limited size. When an application provides an access hint, the possible *benefit* of prefetching that block is weighed against the *cost* of evicting the least valuable block in the cache. However, because TIP only deals with buffer allocation for prefetched data, the cost-benefit analysis is simpler than ours, which must also deal with page allocations of no direct benefit to normal execution.

One aspect of our design that we would like to improve further is memory management. Demke and Mowry [2] demonstrate substantial benefits by proactively evicting pages from memory when memory contention is high. They describe a compile-time technique which deduces the accesses an application will make during execution and inserts *release hints* for blocks that are unlikely to be accessed in the near future. When coupled with their compile-time prefetch hints, a run-time system can approximate Cao's scheduling rules. Unfortunately, static insertion of release hints is limited by difficult interprocedural analysis. We hypothesize that a speculative execution approach may be able to expand the range of applications for which release hints could be automatically generated.

Finally, Chang [9] demonstrates that speculative execution can improve system performance even when concurrent processes contend for processing cycles or disk bandwidth. While our memory control mechanism was designed with multi-process systems in mind (for example, speculative process overheads are measured individually) it has yet to be evaluated in a multi-programming environment.

9 Conclusions

Recent work demonstrated that speculative execution has the potential to greatly improve the performance of I/O-intensive applications, while overcoming the limitations of compiler-assisted prefetching approaches. However, the previous user-level design requires the implementation of a complex, architecture-specific binary modification tool, benefits only explicit-I/O applications that have been transformed by such a tool, and does not limit the overhead incurred by increased memory contention.

In this paper, we present an in-kernel design for capturing the benefits of speculative execution while overcoming these limitations. We demonstrate that, for explicit-I/O applications, a simple design which leverages existing operating system mechanisms delivers benefits comparable to the prior user-level design. We then show that specialized versions of standard OS mechanisms – a fast, preemptible reforking operation and directional copy-on-write – greatly increase the benefits provided to swapping applications. Finally, we demonstrate a mechanism for limiting speculative overhead, while not impeding beneficial speculative execution, by scheduling speculative executions based on their memory impact. Our experience in implementing and evaluating speculative execution within Linux suggests that providing speculative execution within an operating system can be both feasible and effective.

Acknowledgements

We would like to thank Garth Gibson, Khalil Amiri and the anonymous reviewers for providing many helpful comments that greatly improved the paper. We would also like to thank Angela Demke for making her I/O benchmark suite available to us.

References

[1] E. Grochowski. IBM magnetic hard disk drive technology. http://www.almaden.ibm.com/sst/html/leadership/leadership.htm, 2001.

[2] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler inserted releases to manage physical memory intelligently. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[3] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[4] T. Mowry, A. D. Brown, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[5] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[6] D. Bitton, C. Orji, and C. Turbyfill. The AS3AP benchmark. In *Database and Transaction Processing Sys. Performance Handbook*. Morgan Kaufmann, 1991.

[7] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, 1991.

[8] M. Ebling, L. Mummert, and D. Steere. Overcoming the network bottleneck in mobile computing. In *Workshop on Mobile Computing Systems and Applications*, 1994.

[9] F. Chang. Using speculative execution to automatically hide I/O latency. Technical Report CMU-CS-01-172, Carnegie Mellon University, 2001.

[10] R. J. Feiertag and E. I. Organisk. The multics input/output system. In *Proceedings of the 3rd ACM Symposium on Operating Systems Principles*, 1971.

[11] M.K. McKusick, W.J. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[12] D. Kotz and C. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.

[13] T. Madhyastha, G. A. Gibson, and C. Faloutsos. Informed prefetching of collective I/O requests. In *Proceedings of the ACM/IEEE SC99 Conference*, 1999.

[14] C. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.

[15] M. L. Palmer and S.B. Zdonik. FIDO: A cache that learns to fetch. In *Proceedings of the Conference on Very Large Data Bases*, 1991.

[16] K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, 1993.

[17] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

[18] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of the USENIX Winter Technical Conference*, 1997.

[19] T. Kroeger and D. Long. The case for efficient file access pattern modeling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS)*, 1999.

[20] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, 26(10):938–947, 1977.

[21] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1995.

[22] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.