Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999

Automatic I/O Hint Generation through Speculative Execution

Fay Chang Garth A. Gibson

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{fwc,garth}@cs.cmu.edu

Abstract

Aggressive prefetching is an effective technique for reducing the execution times of disk-bound applications; that is, applications that manipulate data too large or too infrequently used to be found in file or disk caches. While automatic prefetching approaches based on static analysis or historical access patterns are effective for some workloads, they are not as effective as manually-driven (programmer-inserted) prefetching for applications with irregular or input-dependent access patterns. In this paper, we propose to exploit whatever processor cycles are left idle while an application is stalled on I/O by using these cycles to dynamically analyze the application and predict its future I/O accesses. Our approach is to speculatively pre-execute the application's code in order to discover and issue hints for its future read accesses. Coupled with an aggressive hint-driven prefetching system, this automatic approach could be applied to arbitrary applications, and should be particularly effective for those with irregular and, up to a point, input-dependent access patterns.

We have designed and implemented a binary modification tool, called "SpecHint", that transforms Digital UNIX application binaries to perform speculative execution and issue hints. TIP [Patterson95], an informed prefetching and caching manager, takes advantage of these application-generated hints to better use the file cache and I/O resources. We evaluate our design and implementation with three real-world, disk-bound applications from the TIP benchmark suite. While our techniques are currently unsophisticated, they perform surprisingly well. Without any manual modifications, we achieve 29%, 69% and 70% reductions in execution time when the data files are striped over four disks, improving performance by the same amount as manually-hinted prefetching for two of our three applications. We examine the performance of our design in a variety of configurations, explaining the circumstances under which it falls short of that achieved when applications were manually modified to issue hints. Through simulation, we also estimate how the performance of our design will be affected by the widening gap between processor and disk speeds.

This research is sponsored by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by an ONR graduate fellowship, and by the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Intel, Quantum, Seagate Technology, Storage Technology, Wind River Systems, 3Com Corporation, Compaq, Data General/Clariion, and Symbios Logic.

1 Introduction

Many applications, ranging from simple text search utilities to complex databases, issue large numbers of file access requests that cannot always be serviced by in-memory caches. Due to the disparity between processor speeds and disk access times, the execution times of these applications are often dominated by I/O latency. Furthermore, since disk access times are improving only slowly, these applications are receiving decreasing benefits from the rapid advance of processor technology, and I/O latency is accounting for an increasing proportion of their execution times.

File systems can automatically hide disk latency during file writes by performing write-behind buffering [Powell77], in which they inform the application that the write request has completed before propagating the data to disk. Automatically hiding the disk latency of file reads is more complicated since, in most applications, the requested data is used as soon as the read returns. *Prefetching*, requesting data before it is needed in order to move it from a high-latency locale (e.g. disk) to a low-latency locale (e.g. memory), is a well-known technique for hiding read latency. To be effective, prefetching requires that the I/O system provide more bandwidth than the application already consumes. Fortunately, we can construct cost-efficient I/O systems capable of providing adequate bandwidth by striping data across an array of disks [Patterson88] or, to facilitate sharing of I/O resources, across multiple higher-level entities like file servers or network disks [Cabrera91, Hartman94, Gibson98].

The difficulty with prefetching lies in knowing how to accurately determine what and when to prefetch. Prefetching consumes processor, cache and I/O resources; if unneeded data is prefetched, or data is prefetched prematurely, I/O requests for more immediately needed data may be delayed and/or more immediately needed data may be displaced from the file cache. One effective alternative is to manually modify applications so that they explicitly control I/O prefetching. Unfortunately, as we will discuss in the next section, this can be a difficult optimization problem for the programmer. Automatic prefetching, however, can significantly reduce execution time without increasing programming effort, provided that the automatic methods are sufficiently accurate, timely and careful with resource usage. In this paper, we present a novel approach to automatic prefetching that is potentially applicable to virtually all disk-bound applications and should be much more effective than existing automatic approaches for disk-bound applications with irregular and input-dependent access patterns.

Our approach arises from the observation that the cycles during which an application is stalled waiting for the I/O system to service a read request are often wasted. This situation occurs commonly both in desktop computing environments and where disk-bound applications are important enough to acquire exclusive use of a high-performance server machine. Even high-performance disk systems currently have at least 10 millisecond access latencies, so that processors may be wasting *millions* of cycles during each I/O stall. We propose that a wide range of disk-bound applications can use these cycles to dynamically discover their own future read accesses by performing *speculative execution*, a possibly erroneous pre-execution of their code.

We present a design for automatically transforming applications to perform speculative execution and issue hints for their future read accesses. Our design takes advantage of TIP [Patterson95], an informed prefetching and caching manager that uses application-generated hints to better exploit the file cache and I/O resources. We have implemented a binary modification tool, SpecHint, that performs this transformation. Using SpecHint, we obtain substantial reductions (29%, 69% and 70%) in the execution times of three real-world applications from the TIP benchmark suite [Patterson95] when the data is striped over four disks. For two of the three applications, we automatically obtain the same benefit as was obtained by manually modifying the applications to issue hints. We examine the performance of our design in a variety of configurations, explaining the circumstances under which it falls short of the performance achieved by manually-hinted prefetching. Through simulation, we also estimate how the performance of our design will be affected by the widening gap between processor and disk speeds.

This paper is organized as follows. In Section 2, we discuss previous prefetching mechanisms. In Section 3, we present our new automatic approach and our design for transforming applications. In Section 4, we describe our experimental framework and results. Finally, in Sections 5, 6, and 7, we present future work, related work, and conclusions.

2 Prefetching background

As mentioned in the introduction, applications can be manually modified to control I/O prefetching. For example, programmers can explicitly separate a request for data from the requirement that the data be available by issuing an asynchronous I/O call. However, there is a serious drawback to using asynchronous I/O. The size of the file cache, the latency and bandwidth of the I/O system, and the level of contention for the file cache and I/O system all affect the ideal scheduling of I/O requests. Issuing an asynchronous read call, however, causes the operating system to immediately issue a disk request for any uncached data specified by the call. Therefore, in redesigning an application to issue asynchronous I/O calls, a programmer implicitly makes assumptions about the characteristics of the systems on which the application will be executed.

Programmers can address this issue by using more sophisticated prefetching mechanisms, e.g. by modifying applications to issue hints for future read requests to a module that considers the dynamic I/O and caching behavior of the system before acting on the hint [Patterson94] (discussed further in Section 2.1). However, this does not avoid the higher-level problems with manual modification. First, manual modification requires that source code be available. Second, manual modification can involve formidable programming effort, both in understanding how the code currently generates read requests and in determining how the code should be modified so that the application will benefit from I/O prefetching. While some applications will only require the insertion of a few lines of code in a few strategic locations, other applications may require significant structural reorganization to support accurate and timely I/O prefetching [Patterson97]. Accordingly, we expect such modifications to be made only by a small fraction of programmers on a small fraction of programs. Therefore, automatic approaches are desirable.

The most widespread form of automatic I/O prefetching is the sequential read-ahead performed by most operating systems [Feiertag71, McKusick84] that exploits the preponderance of sequential whole-file reads [Ousterhout85, Baker91]. However, sequential read-ahead has limited utility when files are small. Furthermore, sequential read-ahead will not help, and may hurt, when access patterns are nonsequential.

In a more sophisticated history-based approach for automating I/O prefetching, the operating system gathers information about past file accesses and uses it to infer future file requests [Kotz91, Curewitz93, Griffioen94, Kroeger96, Lei97]. History-based prefetching is particularly well-suited for discovering and exploiting access patterns that span multiple applications. For example, it may implicitly recognize the edit-compile-run cycle and prefetch the appropriate compiler, object files, or libraries while a user is editing a source file. When applied to disk-bound applications such as those used in our experiments, however, history-based approaches are less appropriate. These approaches are inherently limited by the tradeoff between the amount of history information retained and the achievable resolution in prefetching decisions. High resolution prediction – the ability to anticipate irregular block accesses in long-running disk-bound applications, for example – could require prohibitively large traces of prior executions. By whatever measures a particular history-based prefetching system reduces the amount of information it retains – e.g. by tracking only certain types of events or only the most frequently occurring events – the system will also sacrifice its ability to predict the accesses of applications whose access patterns vary widely between runs and/or applications that heavily exercise the I/O system but recur infrequently.

For these types of applications, we need a different approach for automating I/O prefetching. We would like an approach that considers precisely the factors which determine a specific application's stream of read requests, without burdening the operating system by requiring it to maintain long-term application-specific information. One such approach is for a tool, generally a compiler, to statically analyze an application in order to determine how read requests will be generated, and then transform the application so that the appropriate I/O prefetching will occur [Mowry96, Trivedi79, Cormen94, Thakur94, Paleczny95]. Such static approaches have proven extremely effective at reducing execution times for loop-intensive, array-based applications. However, these approaches are limited by hard interprocedural static analysis problems, especially because I/O is often an "outer loop" activity separated from the core computation by many layers of abstraction (procedure calls and jump tables, for example).

Our approach is based on having applications perform speculative execution, which is essentially a form of dynamic self-analysis. As with static approaches, we are able to capture application-specific factors which are expensive for history-based prefetching systems to extract and retain. Unlike static approaches, however, we do not require detailed understanding of the control and data flow of the application. Instead, our approach requires only a few simple static analyses and transformations. In addition, by relying on dynamic analysis, our approach can easily take advantage of input data values as they becomes available during the course of execution.

2.1 TIP

In the last section, we discussed why prefetching and caching decisions should depend on the dynamic state of the system. Patterson [Patterson94] and Cao [Cao94] have argued that this issue should be addressed by separating access understanding from resource allocation. Specifically, Patterson proposed that applications issue informing hints that disclose their future accesses as a sequence, allowing the underlying system to make optimal global decisions about what and when to prefetch, and what to eject from memory to make space for prefetched data. By issuing informing hints, applications would be both portable to other machines and sensitive to the changing conditions on any given machine.

Benchmark	Improvement	Description
`Agrep`	72%	text search
`Gnuld`	66%	object code linker
`XDataSlice`	70%	scientific visualization
`Davidson`	12%	computational physics
`Postgres, 20%`	48%	database join,
`Postgres, 80%`	69%	% tuples resulting
`Sphinx`	21%	speech recognition

Table 1: *Reductions in execution times using applications* ***manually modified*** *to issue hints for future accesses, as reported by Patterson [Patterson97]. These results were obtained on a 175MHz Digital 3000/600 with 128MB of memory running Digital UNIX 3.2c when the data was striped over four HP2247 disks with a 64KB striping unit.*

To validate his proposal, Patterson designed and built TIP, an informed prefetching and caching manager that replaces the Unified Buffer Cache manager in the Digital UNIX 3.2 kernel. TIP attempts to improve use of the file cache and I/O resources by performing a cost-benefit analysis. Roughly speaking, TIP estimates the benefit of prefetching in response to a hint based on the accuracy of previous hints from the application and the immediacy of the hint. It balances this estimated benefit against an estimated cost of prefetching, which is composed of the estimated cost of ejecting a block from the cache and the estimated opportunity cost of using the I/O system. On a benchmark suite that included a range of applications, informed prefetching and caching reduced execution times by 12-72% when data files were striped over four disks (see Table 1), clearly demonstrating that application-level hints for future read accesses can be effectively used to guide intelligent prefetching and caching decisions that take advantage of the bandwidth provided by a parallel I/O system.

These results are impressive, but the applications had to be manually modified to issue hints. For some of the applications, such as Gnuld and Sphinx, this involved significantly restructuring the code so that hints could be issued earlier and obtain more benefit from prefetching. The purpose of our research is to make the demonstrated benefits of prefetching readily accessible by automating the generation of informing hints.

Our design and implementation of speculative execution for automatic hint generation assumes that TIP is the underlying prefetching system (but could be retargetted to other prefetching systems). As shown in Table

Ioctl	Parameters	Description
TIPIO_SEG	batch of (filename, offset, length)	hints one or more segments from a named file
TIPIO_FD_SEG	batch of (file descriptor, offset, length)	hints one or more segments from an open file
TIPIO_CANCEL_ALL	none	cancels all outstanding hints from the issuing process

Table 2: *Relevant portion of the hinting interface exported by TIP. We do not exercise the capability for batching hints as speculative execution discovers reads one at a time. Recall that the standard UNIX read call takes a file descriptor, a pointer to a buffer, and a length as its parameters.*

2, TIP's hint interface includes calls which are almost directly analogous to the basic UNIX read calls. Our only modification of TIP was the addition of a CANCEL_ALL_HINTS call, which was accomplished with a few lines of code. The CANCEL_ALL_HINTS call will only cancel hints; once issued, prefetch requests cannot be cancelled.

3 Speculative execution

We propose that applications continue executing speculatively after they have issued a read request that misses in the file cache; that is, when they would ordinarily stall waiting for a disk read to complete. During this speculative execution, applications should issue the appropriate (non-blocking) hint call whenever they encounter a read request in order to inform the underlying prefetching system that the data specified by that request may soon be required. If the hinted data is not already cached and the prefetching system believes that prefetching the hinted data is the best use of disk and cache resources, then it should issue an I/O request for the hinted data. If the I/O system can parallelize fetching hinted data with its servicing of the outstanding read request, then the latency of fetching the data may be partially or completely hidden from the application.

Figure 1 depicts the intuition as to why speculative execution works. Consider an application which issues four read requests for uncached data and processes for a million cycles before each of these read requests. Assume that the data is distributed over three disks, that the disk access latency is three million cycles, and that there are sufficient cache resources to store all of the data used by this application once fetched. If we assume that speculative execution proceeds at the same pace as normal execution, then, while normal execution is stalled waiting for the first read request to complete, speculative execution may be able to issue hints for the remaining three read requests. If the data layout allows the hinted data to be fetched in parallel with service of the outstanding read request and the subsequent processing, then all of the subsequent read requests will hit in the cache, and the application's execution time will be more than halved.

Of course this is an oversimplification. Speculative execution will incur some run-time overhead. In addition, the pre-execution may be incorrect because some of the data values used during speculation may be incorrect (for example, those in the buffer into which data for

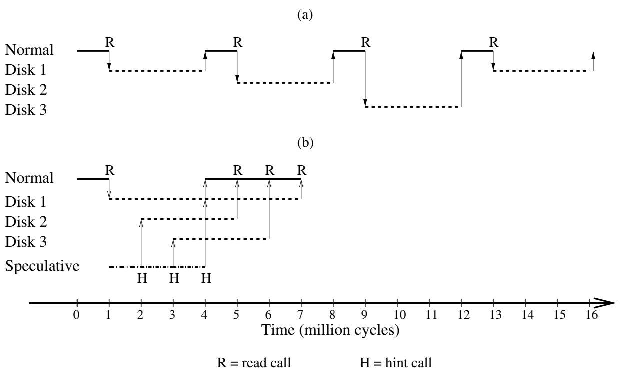

Figure 1: *Simplified example of how speculative execution reduces stall time: (a) shows how execution would normally proceed for a hypothetical application, and (b) shows how execution might proceed for the application if it performs speculative execution during I/O stalls in order to generate I/O hints. Performing speculative execution could more than halve the execution time of this example.*

the outstanding read request is being placed). Incorrect hints may lead the prefetching system to make erroneous prefetching and caching decisions. For example, they may result in the disks being busy reading unneeded data instead of servicing requests that are stalling the application, in keeping data in the cache that will not be needed but was identified by an incorrect hint, or in ejecting data from the cache that will be needed but was not identified by a hint. Furthermore, performing speculative execution will increase contention for other machine resources. This may result in normal (non-speculative) execution experiencing additional page faults, TLB misses and/or processor cache misses. Finally, if there is contention for the processor or the I/O system as, for example, with a multithreaded server or in a multiprogrammed environment, then speculative execution will have less opportunity to improve performance.

3.1 Design goals

We identify three basic design goals for how applications should be transformed to use speculative execution. Specifically, the transformation should be:

- *Correct* – the results of executing a transformed application should match those of executing the original application;
- *Free* – a transformed application should, at worst, be slower than the original application by an insignificant amount; and

- *Effective* – as many as possible of the application's requests for uncached data should be hinted in a timely fashion, with the minimum possible impact on machine resources.

3.2 Our design

Our design currently requires no specialized operating system support (other than the prefetching system and strictly prioritized kernel threads) and is appropriate for single-threaded applications. The basic element in our current design is the addition of a new kernel thread to the application. We call this thread the *speculating thread*, and its purpose is to perform speculative execution while the "original" application thread is stalled. We ensure that the speculating thread only executes when the original thread is stalled by assigning the speculating thread a low priority and selecting a preemptive scheduling policy which time-slices amongst only the highest priority runnable threads. A hint call is issued by the speculating thread whenever it encounters a read call.

3.2.1 Ensuring program correctness

There are three ways in which performing speculative execution could potentially change the behavior of the application. First, since the speculating thread shares an address space with the original thread, it could distort normal execution by changing code or data values that will be used by the original thread. Second, the speculating thread could produce side-effects visible outside the process, changing the impact of the application on the system. Finally, the speculating thread may inadvertently use inappropriate data values, like dividing by 0 or accessing an illegal address, that disrupt the execution of the application.

We ensure the correctness of our transformation by avoiding these potential problems. We prevent the speculating thread from producing side-effects visible outside the process by not allowing the speculating thread to issue any system calls except the hint calls (described in Table 2), and the `fstat()` and `sbrk()` calls.[1] We prevent the use of inappropriate data values from disturbing normal execution by installing signal handlers to catch any exceptions generated by the speculating thread, halting speculative execution until the original thread blocks on a new read call. Finally, we prevent the speculating thread from changing code or data values used by the original thread through *software-enforced copy-on-write*.

Inspired by software fault isolation [Wahbe93], software-enforced copy-on-write involves adding checks before each load and store instruction executed by the speculating thread, and adding a data structure to keep track of which memory regions have been copied and where their copies reside. Before each store instruction executed by the speculating thread, a check is added which accesses the data structure to discover whether the targetted memory region has already been copied. If so, the store is redirected to access the copy. If not, the memory region is copied, the data structure is updated, and the store is redirected to the newly created copy. Similarly, before each load instruction, a check is added which accesses the data structure to discover whether the referenced memory region has already been copied and, if so, redirects the load to obtain the value stored in the copy, which is the "current" value with respect to speculative execution.

Since load and store instructions comprise approximately 30% of the average instruction mix, software-enforced copy-on-write could be an expensive solution. For example, it may appear that the original thread would need to execute many additional branching instructions to avoid performing the checks. We avoid this overhead by making a complete copy of the binary's text section and constraining the speculating thread to only execute within the copy, which we call the *shadow code*. This permits us to add copy-on-write checks only around loads and stores in the shadow code, so that the original thread does not need to execute any additional instructions to support software-enforced copy-on-write.

Minimizing additional instructions in the original thread's code path is an example of our effort to minimize the *observable overhead* of supporting speculative execution. The checking necessary to perform software-enforced copy-on-write does not add directly to the execution time of the application; it simply causes speculative execution to proceed more slowly than normal execution; that is, it is *nonobservable overhead*. In general, we prefer design choices that incur nonobservable overhead to those that incur observable overhead since they seem less likely to affect worst-case performance.

We ensure that the speculating thread only executes shadow code by statically and/or dynamically checking and redirecting all control transfers (that is, possibilities for non-sequential changes in execution address). All control transfers that can be statically resolved are statically redirected to the appropriate address in the shadow code. Control transfers that cannot be statically resolved include those dynamically calculated using jump tables, corresponding to switch statements. Our binary modification tool only recognizes a few of the possible compiler-dependent jump table formats, so it can only statically handle switch statement control transfers that rely on jump tables in a recognized format. All other control transfers are statically redirected to call a special

[1] We add a set of memory allocation routines for use by the speculating thread to prevent speculative execution from introducing memory leaks. Notice that the behavior of an application could be inadvertently altered if it depends on its dynamic state (e.g. on the location of its sbrk pointer) or on the last access time of a file. We expect these types of applications to be uncommon.

handling routine with the originally intended target address as an argument. During runtime, if the originally intended target address is in the shadow code, the handling routine allows the speculating thread to proceed to that address. If the address is not in the shadow code but can be mapped to an address in the shadow code, then the handling routine redirects the speculating thread.[2] Otherwise, the handling routine simply prevents the speculating thread from leaving the shadow code (by preventing further progress until a new speculation is started, as discussed in the next section). Notice that, for applications with self-modifying code, this scheme will not allow the speculating thread to execute any newly created code, or to modify the existing shadow code.

One potential advantage of using software-enforced copy-on-write is the flexibility it permits in choosing the size of copy-on-write memory regions. However, when we explored this flexibility by varying the copy-on-write region size from 128B to 8192B, we discovered that it generally made no significant difference to the performance improvements obtained – the only difference larger than 5% was a 9% reduction in performance for Gnuld with a region size of 8192B. All of the results presented in this paper were obtained using 1024B regions.

3.2.2 Generating correct and timely hints

We would like to issue hints for as many of the read calls as possible so that TIP will have as much information as possible on which to base its prefetching and caching decisions. In addition, we would like to issue these hints as early as possible so that there will be ample opportunity to hide the latency of any prefetches. There are two situations that could obstruct these goals. First, because the speculating thread is only allowed to execute when the original thread is blocked, speculative execution could fall "behind" normal execution. If speculation is allowed to proceed in this situation, speculative execution would need to waste many cycles catching up to normal execution before it would be able to issue useful hints (that is, hints for read calls that have not already been issued). For some applications, including those with a long intermediate processing phase, speculative execution might never be able to catch up to normal execution. Second, because the speculating thread proceeds with incomplete state information, speculative execution could "stray" from the execution path that will be taken during normal execution. If speculation is allowed to proceed in this situation, the speculating thread might not be able to hint any future read calls. Even worse, it might generate a stream of incorrect hints, which could significantly hurt performance as explained at the beginning of Section 3. We describe speculative execution as being *on track* if the next hint issued would correctly predict the next unhinted future read call; otherwise, we describe speculative execution as being *off track*. We attempt to keep speculative execution on track as much as possible in order to increase the benefit we will be able to obtain through prefetching.

A pessimistic approach to keeping speculative execution on track would be to restart speculation every time the original thread blocks on a read call, where "restarting speculation" means causing the speculating thread to execute as if it had just returned from the call on which the original thread is currently blocked. However, this bounds how far speculative execution can predict the future to the distance it can progress during a single I/O stall, unnecessarily limiting the potential benefit of speculative execution. We attempt to increase the number of correct and timely hints generated by having the speculating and original threads cooperate to restart speculation only when they detect that speculative execution is off track.

Detecting when speculative execution is off track is accomplished by having the speculating thread record the hints it issues in a new data structure, called the *hint log*. The original thread maintains an index into the hint log and, whenever it is about to issue a read request, it checks the next entry in the hint log. If there is no next entry in the hint log, then the original thread knows that speculative execution is behind normal execution and is therefore off track. If there is an entry but it does not match the read request, then the original thread knows that speculative execution strayed from the correct execution path at some point in the past and is therefore off track. On the other hand, if the next entry matches the read, then, as far as the original thread can determine, speculative execution may still be on track.

Upon detecting that speculative execution is off track, the speculating and original threads also cooperate to restart speculation. In order to restart speculation, the speculating thread needs the original thread's state. When the original thread detects that speculative execution is off track, it copies the values of its registers into a data structure since the speculating thread cannot otherwise acquire their values and sets a "restart" flag to inform the speculating thread that it is off track. This work is performed before the original thread issues its read request because, if the original thread blocks on the read request, the speculating thread will have the opportunity to run. The speculating thread polls the restart flag frequently and, if the flag is set, cleans up its current speculation by cancelling any outstanding hints and clearing the copy-on-write data structure. The speculating thread then restarts speculation by loading the original thread's saved register values, making a copy of the

[2] Currently, the handling routine can only map function addresses, so that it can redirect control transfers through function pointers, but not computed goto statements.

original thread's stack[3], and jumping to the instruction which immediately follows the read system call in the shadow code.

Through this cooperation, we ensure that the speculating thread will not waste many cycles executing behind the original thread. We also ensure that the speculating thread will not waste cycles restarting speculative execution unless there is reason to believe that it is off track. While we cannot ensure that the speculating thread will not perform incorrect speculation and issue erroneous hints, we address this situation when it is detected by the original thread. Finally, we require the original thread to perform little additional work (at most, checking an entry in the hint log and saving its registers once per read) so that observable overhead is small.

3.3 Transforming applications

We use binary modification to automatically transform applications so that they will perform speculative execution. We chose to use binary modification because it does not require source code and can be both language- and compiler-independent. Of course, the speculative execution transformations could also be performed within a compiler.

The SpecHint tool is implemented in 16,000 lines of C code. Currently, the tool is relatively unsophisticated. It is restricted to Digital UNIX 3.2 Alpha binaries produced by the native `cc` compiler that are single-threaded, statically linked, and retain their relocation information. It does not yet perform any loop optimizations, which could significantly decrease the number of copy-on-write checks in some codes. The tool does recognize, and remove from the shadow code, calls to a few of the standard library output routines (`printf`, `fprintf` and `flsbuf`) because these routines are known not to influence future read accesses and can require many cycles to execute.

As illustrated in Figure 2, the application object files and libraries are first linked with the SpecHint auxiliary object files and the necessary libraries to support threading. The resulting binary is transformed by SpecHint, then linked normally to produces a transformed application executable. The SpecHint object files, which were generated from 4,000 lines of assembly code, include the dynamic memory allocation routines used by the speculating thread and the routine that handles control transfers that cannot be statically resolved (discussed in Section 3.2.1), as well as a routine that the speculating thread executes in order to restart speculation (discussed in Section 3.2.2). They also contain versions of `strncpy` and

[3] In combination with placing dynamic checks on instructions which modify the stack pointer and cannot be statically checked, copying the stack also allows us to avoid copy-on-write checks for load and store instructions off the stack pointer.

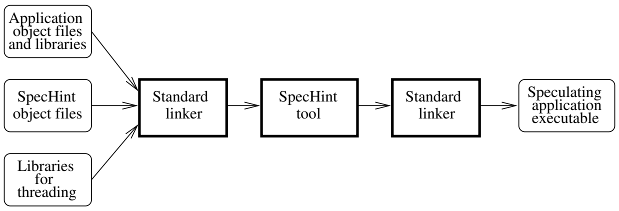

Figure 2: *Transforming applications to use speculative execution. The SpecHint object files contain various routines executed by the original or speculating thread in order to support speculative execution.*

`memcpy` for the shadow code that were hand-optimized to simulate the effect of performing loop optimizations to minimize copy-on-write checks in these standard library routines.

4 Experimental evaluation

In this section, we describe our experimental environment, our benchmarks, and our results. The SpecHint tool implements the design described in the previous section by modifying Alpha binaries for Digital UNIX 3.2. Threading to support speculative execution was implemented using Digital UNIX's POSIX-compliant pthreads library.

Our experiments were conducted on an AlphaStation 255 (233MHz processor) with 256MB of main memory running Digital UNIX 3.2g, where the standard Unified Buffer Cache (UBC) manager was replaced with the TIP informed prefetching and caching manager. To facilitate comparison with Patterson's work [Patterson95], the file cache size was set to 12MB. The automatic read-ahead policy, which was invoked by all unhinted read calls, prefetches approximately the same number of blocks as have been sequentially read, up to a maximum of 64 blocks. The I/O system consisted of four HP C2247 disks (15ms average access times) attached by fast-wide-differential SCSI. Data files were striped over these disks by a striping pseudodevice with a striping unit of 64KB. A new file system was created to hold the files used in our experiments. All tests were run with the file cache initially empty. All reported results are averages over five runs. To facilitate comparison with programmer-inserted hints, we reran the manually modified applications [Patterson95] on this testbed.

4.1 Benchmark applications

We evaluated the effectiveness of our approach on three benchmark applications from the TIP benchmark suite [Patterson97].

Agrep (version 2.04) is a fast full-text pattern matching program. The application loops through the files specified on its command line, opening and reading each file sequentially. Therefore, the arguments to Agrep completely specify the stream of read accesses it will perform. In the benchmark, Agrep searches 1349 Digital

Benchmark	Modification time (s)	Transformed executable size	% increase in size
`Agrep`	21	1,648 KB	610%
`Gnuld`	23	2,408 KB	349%
`XDataSlice`	151	10,792 KB	138%

Table 3: *Transformed application statistics.*

UNIX kernel source files occupying 2928 disk blocks for a simple string that does not occur in any of the files.

Gnuld (version 2.5.2) is the Free Software Foundation's object code linker. The input object files are specified on the command line. Gnuld first reads each object file's file header, symbol header, symbol tables and string tables. The location of each file's symbol header is stored in its file header, and the locations of its symbol and string tables are stored in its symbol header. Gnuld then makes up to nine small, non-sequential reads in each object file to gather debugging information. The locations of these reads are determined from the symbol tables. Finally, Gnuld loops through the different non-debugging sections that appear in an object file, reading the corresponding section from each of the object files. Interspersed with the reads, Gnuld processes the data in order to produce and output an executable. In the benchmark, 562 binaries are linked to produce a Digital UNIX kernel.

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 Patterson modified the application to load data dynamically from large data sets [Patterson95]. In the benchmark, XDataSlice retrieves 25 random slices (the same slices used for Patterson's experiments) through a data set of 512^3 32-bit floating-point numbers that resides in 512MB of disk space.

4.2 Transformed applications

The application binaries were transformed by SpecHint on a 500MHz AlphaStation 500 with 1.5GB of memory. SpecHint is an unoptimized research prototype. Nevertheless, as shown in Table 3, SpecHint was able to modify our benchmark applications in a reasonable amount of time, 21 to 151 seconds. The resulting binaries were processed by the standard linker to produce speculating executables that, unlike the original application executables, contain shadow code, the SpecHint binaries, and libraries to support threading. These additions resulted in a 138% to 610% increase in executable size.

4.3 Overall performance results

As shown in Figure 3, performing speculative execution significantly reduces the execution times of our benchmark applications (by 69%, 29% and 70%, respectively, for Agrep, Gnuld and XDataSlice). For Agrep and XDataSlice, we were able to automatically achieve the same performance improvements obtained when the applications were manually modified to issue hints. For Gnuld, our gain was much less than that of the manually modified application, but still represents a substantial improvement over the original non-hinting application. Based on these results, and considering the relative unsophistication of our tool, speculative execution promises to be an effective technique for exploiting disk parallelism and underutilized processor cycles to reduce the execution time of disk-bound applications.

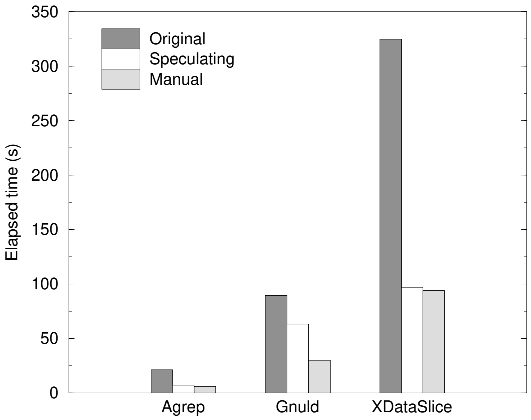

Figure 3: *Performance improvement.* Original *corresponds to the original, non-hinting applications;* Speculating *corresponds to the applications transformed to perform speculative execution for hint generation; and* Manual *corresponds to the applications manually modified to issue hints. In all cases, the non-hinted read calls issued by the applications invoked the operating system's sequential read-ahead policy (which is described at the beginning of Section 4).*

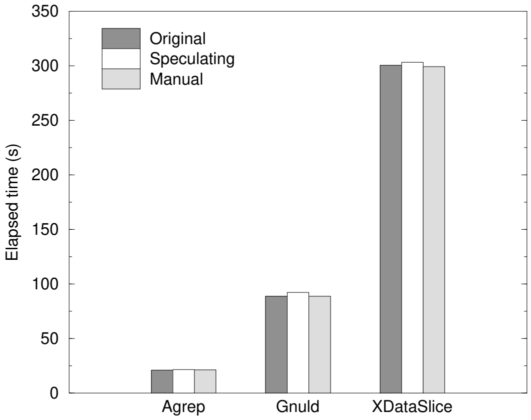

Figure 4: *Runtime overhead of supporting speculative execution, as captured by running the benchmarks with TIP configured to ignore hints.*

If TIP is configured to ignore hints, the applications that perform speculative execution were no more than 4%, and as little as 1%, slower than the original applications as shown in Figure 4. These figures capture all of the factors that can contribute to the worst-case perfor-

Benchmark		Read calls	Read blocks	Read bytes	Write calls	Write blocks	Write bytes
`Agrep`	total	4,277	2,928	18,091,527	0	0	0
	% hinted	68.1%	99.6%	99.7%	–	–	–
	inaccurately hinted	0	0	0	–	–	–
	% manually hinted	*68.3%*	*99.8%*	*>99.9%*	–	–	–
`Gnuld`	total	13,037	20,091	60,158,290	2343	3418	8,824,188
	% hinted	54.9%	67.5%	89.7%	–	–	–
	inaccurately hinted	2,336	6,721	37,177,440	–	–	–
	% manually hinted	*78.4%*	*86.0%*	*99.6%*	–	–	–
`XDataSlice`	total	46,356	46,352	370,663,914	2	2	4081
	% hinted	97.5%	97.5%	99.9%	–	–	–
	inaccurately hinted	0	0	0	–	–	–
	% manually hinted	*97.6%*	*97.6%*	*>99.9%*	–	–	–

Table 4: *Hinting statistics.* Total *includes explicit file calls only. The hinting behavior of the speculating applications is described by the* % hinted *and* inaccurately hinted *figures, and can be compared with the behavior of the manually modified applications (which issued no inaccurate hints) described by the* % manually hinted *figures. The number of read calls is sometimes larger than the number of read blocks because, for example, Agrep issues at least one extra read call per file to detect the end of the file. Discounting these non-data-returning reads (which do not need to be hinted), over 99% of Agrep's read calls were hinted.*

mance of the speculating applications except the potential negative effects of any erroneous hints. These factors include increased memory contention, the overhead of checking hint log entries before issuing read calls, and the overhead of executing an initialization routine that, among other things, spawns the speculating thread.

4.4 Performance analysis

Having established that speculative execution achieves significant performance improvements, we examine the behavior of the speculating applications and attempt to explain the differences between our results and those obtained with manually modified applications.

The primary metric for automatic hint generation is the number of correct hints generated. Table 4 summarizes the hinting behavior of the original and transformed applications. For Agrep and XDataSlice, we found that speculative execution was able to issue hints for nearly as many of the read calls as the manually modified applications. However, speculative execution was significantly less successful for Gnuld, hinting only 55% of the read calls in contrast to the 78% that the manually modified application was able to hint.

There are two basic reasons why speculating applications may hint fewer read calls than manually modified applications. One is that speculating applications must determine what to hint dynamically, but are only allowed to pursue hint discovery while normal execution is stalled. In fact, the more successfully a speculating application generates hints that will hide I/O latency, the less opportunity it will have to pursue hint discovery, unless the application is bandwidth-bound. The other reason is that data dependencies limit how early prefetches can be issued. For example, if the data specified by the next read call depends on the data returned by the currently outstanding read call, then speculative execution will not be able to hint the next read call.

Agrep is the most likely of our applications to be affected by the fact that hint discovery is only performed during I/O stalls. Agrep has the largest median number of cycles between read calls – 30362, 15902 and 4454 for Agrep, Gnuld and XDataSlice, respectively. It also has the largest ratio between the median number of cycles between hint calls and the median number of cycles between read calls – 7.5, 1.6 and 1.3 for Agrep, Gnuld and XDataSlice, respectively. (This ratio, which we call the *dilation factor*, is larger than one mainly due to the copy-on-write checks performed during speculative execution.) Accordingly, of our three applications, the speculating Agrep generates hints at by far the slowest rate. However, the almost equal gains achieved by the speculating Agrep and the manually modified Agrep indicate that this property of our design has negligible impact.

During the process of manually modifying an application to issue hints, programmers can make the application more amenable to prefetching by restructuring the code to increase the number of cycles between dependent read calls. As mentioned in Section 2.2, this was the case for the manually modified Gnuld. The speculating Gnuld, however, was produced from the original, unmodified code. It is only able to hint 55% of the read calls because a speculating application cannot hint a read call if it depends on a prior read and there are no I/O stalls between when the prior read completes and when the read call is issued. In addition, since a read cannot be hinted until all the data it is dependent on becomes available, data dependencies may cause hints to be issued too late to fully hide the latency of fetching the specified data. Comparing the speculating Gnuld to the manually modified Gnuld, over five times as many data blocks were only partially prefetched before being requested by the application (as shown in the *Partially* column of Table 5), indicating that the speculating Gnuld experienced many more I/O stalls. Finally, since each speculation proceeds with the assumption that future read calls are not data dependent, data dependencies may cause erroneous hints

Benchmark		Cache Block Reads	Prefetched Blocks	Prefetched Blocks Fully	%	Partially	%	Unused	%	Cache Block Reuses
`Agrep`	Original	3,424	1,031	529	51.3%	499	48.4%	3	0.4%	416
	SpecHint	3,726	3,003	2,707	90.2%	272	9.1%	23	0.8%	655
	Manual	3,423	2,947	2,687	91.2%	258	8.8%	1	0.0%	421
`Gnuld`	Original	24,074	5,511	2,544	46.2%	2,014	36.6%	952	17.3%	12,435
	SpecHint	25,353	12,855	3,498	27.2%	5,432	42.3%	3,924	30.5%	13,646
	Manual	23,892	10,018	8,933	89.2%	1,057	10.6%	27	0.3%	13,519
`XDataSlice`	Original	49,997	60,702	12,806	21.1%	12,664	20.9%	35,231	58.0%	4,162
	SpecHint	50,810	45,338	40,319	88.9%	4,907	10.8%	112	0.3%	4,973
	Manual	49,782	44,938	40,167	89.4%	4,750	10.6%	20	0.0%	4,491

Table 5: *Prefetching and caching statistics. For the original, non-hinting applications, the prefetching figures are the result of the operating system's sequential read-ahead policy. For the speculating applications, the prefetching figures also include TIP's hint-driven prefetching.* Cache Block Reads *is the number of block reads from the file cache.* Prefetched Blocks *is the number of blocks prefetched from disk.* Fully *is the number of blocks whose prefetch completed before being requested by the application,* Partially *is the number of blocks partially prefetched before being requested by the application, and* Unused *is the number of prefetched blocks that were not accessed by the application before being ejected from the file cache. A* Cache Block Reuse *is counted each time a cached block services a second or subsequent request, and therefore indicates the effectiveness of caching. The closeness of the* Cache Block Reuse *figures indicates that erroneous prefetching did not significantly harm caching behavior.*

to be generated. The speculating Gnuld generates 2,336 erroneous hints, as shown in Table 4, contributing to the prefetching of 3,924 unused data blocks, as shown in Table 5.

Prefetching speculatively, and therefore sometimes incorrectly, is not new. History-based mechanisms all have this property. Specifically, Digital UNIX has an aggressive automatic read-ahead policy based on the expectation that files are read sequentially. It prefetches approximately the same number of blocks as have been read sequentially, up to a maximum of 64 blocks. For applications that issue nonsequential reads to large files, like XDataSlice, this read-ahead policy can be entirely too aggressive. As shown in Table 5, 58% of the blocks prefetched by sequential read-ahead for the non-hinting XDataSlice are not used. In contrast, since the read-ahead policy is only invoked by unhinted read calls and the hinting XDataSlices generate hints for almost all of the read calls, the hinting XDataSlices are able to almost eliminate the erroneous prefetches generated by the read-ahead policy.

4.5 Performance side-effects

In addition to generating hints, speculative execution will have other, less desirable performance effects. For example, since the speculating thread uses shadow code and performs copy-on-write, the speculating applications have larger memory footprints, consume memory more rapidly, and experience more page faults than the original applications. Table 6 shows that the memory footprints increase by 544 KB to 4.1 MB, the number of page reclaimes increases by 95 to 633, and the number of page faults increases by 12 to 40. In addition, the speculating applications may generate extraneous signals because speculative execution may use erroneous data in its calculations. Table 6 shows that the speculating applications generate up to 39 extraneous signals. However,

Benchmark		Footprint	Reclaims	Faults	Sigs
`Agrep`	Original	160 KB	39	4	0
	SpecHint	704 KB	134	16	0
	Manual	152 KB	39	4	0
`Gnuld`	Original	10.1 MB	1,341	12	0
	SpecHint	14.2 MB	1,974	52	39
	Manual	10.5 MB	1,389	14	0
`XDS`	Original	62.0 MB	8,105	61	0
	SpecHint	62.5 MB	8,202	93	2
	Manual	62.1 MB	8,104	60	0

Table 6: *Performance side-effects of speculative execution.* Footprint *is the maximum amount of memory that is physically mapped on behalf of the application at any time.* Reclaims *is the number of page reclaims, and* Faults *is the number of page faults, generated by the application. A page reclaim occurs if a referenced page is still in memory but is not physically mapped, and therefore requires operating system intervention but does not require a disk access. On our evaluation platform, at least one third of the memory-resident pages are not physically mapped, as determined by an LRU policy.* Sigs *is the number of signals generated by the application. For our applications, these signals were either segmentation violations or floating point exceptions.*

many of the additional page reclaims and page faults, and all of the additional signals, will occur while the original thread is blocked on I/O, so that they would be nonobservable overhead. As described in Section 4.3, the observable overhead of these performance side-effects is captured within the less than 4% increases in runtime observed when hints were disabled.

4.6 Varying file cache size

All previously reported results for the manually modified applications were obtained with a 12 MB file cache [Patterson95, Patterson97]. We measure the sensitivity of our results to the file cache size by running the benchmarks with a smaller (6 MB) file cache, and a larger (64MB) file cache. The cache size can affect performance because the sequential read-ahead policy sometimes prefetches data that will be accessed much later, and larger cache sizes may allow more of this data to remain in memory until the future access. For exam-

Benchmark		File cache size		
		6 MB	12 MB	64 MB
Agrep	Original	21.3	21.4	21.2
	SpecHint	6.5 (69%)	6.5 (70%)	6.4 (70%)
	Manual	6.3 (70%)	6.2 (71%)	6.1 (71%)
Gnuld	Original	106.3	89.5	56.5
	SpecHint	74.7 (30%)	63.3 (29%)	45.2 (20%)
	Manual	34.4 (68%)	30.2 (66%)	25.4 (55%)
XDS	Original	295.0	324.6	279.0
	SpecHint	94.6 (68%)	97.0 (70%)	87.8 (69%)
	Manual	91.4 (69%)	94.1 (71%)	85.8 (69%)

Table 7: *Elapsed time of applications as the file cache size is varied (in seconds). Percentages indicate performance improvement relative to the original, non-hinting application.*

ple, as shown in Table 7, the performance of the original, non-hinting Gnuld improves significantly as the cache size increases, reducing the benefit that can be obtained through prefetching. The speculating Gnuld achieves relatively less benefit with a 64MB cache because many of the read calls which it can generate hints for no longer require prefetching, whereas many of the read calls it is unable to hint continue causing I/O stalls. For Agrep and XDataSlice, there is little data reuse and sequential read-ahead seldom fetches data that is accessed much later, so the cache size does not affect the benefit obtained by the hinting applications.

4.7 Varying available I/O parallelism

While four-disk arrays are widely available, we also tested a single disk configuration and smaller and larger arrays. As shown in Table 8, the original, non-hinting applications are unable to derive much benefit from additional disks.

As shown in Figure 5, all the benchmarks receive significantly less benefit from speculative execution when there is only one disk because prefetching can only be overlapped with computation. The performance of speculating Gnuld degrades with one disk because erroneous prefetches consume scarce bandwidth, delaying service for the application's demand requests. As we discuss in Section 5, we believe that simple mechanisms can be employed to address this problem.

One objection to our assumptions – that disk-bound applications will be running on machines that have both disk arrays and no competing tasks to run on the processor – is that more than one disk is attached to a machine only if it is a shared server. However, Rochberg has shown that the TIP system can be effectively extended to allow clients to prefetch from distributed file servers with multiple disks [Rochberg97]. It is these "personal" clients that will be most rich in excess processor cycles.

As shown in Figure 5, the benefit of the hinting applications increase, and their runtimes decrease, when I/O parallelism is available. The benefit obtained by the manually modified applications increases monotonically

Benchmark	Number of Disks			
	1	2	4	10
Agrep	23.8	24.1	21.4	20.1
Gnuld	93.7	101.3	89.5	82.8
XDS	303.5	292.0	324.6	265.7

Table 8: *Elapsed time of original, non-hinting applications as the number of disks is varied (in seconds).*

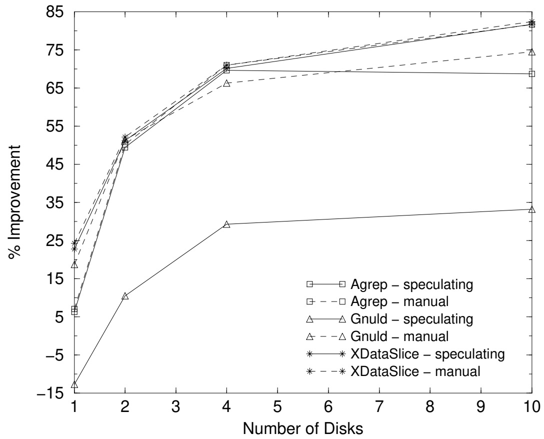

Figure 5: *Performance improvement as the number of disks is varied.*

with the number of disks since these applications always issue enough hints to take advantage of the additional disks. For Agrep, the benefit of the speculating application mirrors that of the manually modified application for the 2 and 4 disk configurations. However, due to the dilation factor discussed in Section 4.4, speculative execution is not far enough ahead of normal execution to issue sufficient hints to keep 10 disks busy. For Gnuld, data dependencies limit hint generation, and therefore the degree to which the speculating application is able to utilize additional disks. For XDataSlice, however, speculative execution generates more than enough hints to take advantage of the additional I/O parallelism.

4.8 Increasing relative processor speed

Due to rapid improvements in processor technology, the gap between processor speeds and I/O latency continues to widen. This will increase the number of cycles per I/O stall, and therefore the progress that speculative execution can make during a single stall. To predict the impact of this trend on the effectiveness of our approach, we modified the striping pseudodevice to delay notification of completed I/O requests. For example, to simulate the effect of doubling the gap between processor and disk speeds, we doubled the time before the system was notified that each I/O request had completed, then scaled our resulting measurements by half.[4] Since disk positioning times and data rates improve at different rates, and

[4]To obtain the desired effect on the perceived service time of prefetch requests, we configured the pseudodevice to limit the number of prefetch requests outstanding at each disk to at most one.

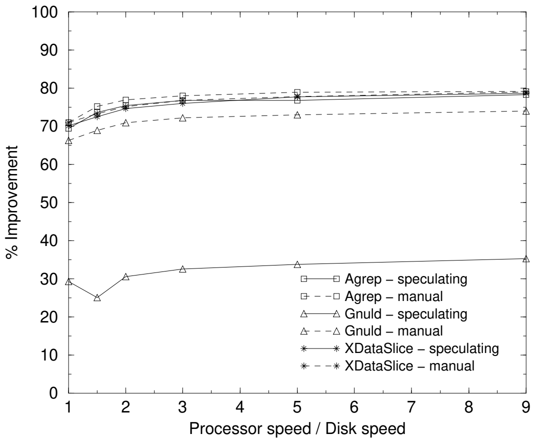

Figure 6: *Results from simulating a widening of the gap between processor and disk speeds. A processor/disk speed ratio of 1 indicates results in our current experimental environment.*

data rates have been improving at 40% per year lately, this simulates an artificially slow transfer rate. However, since the disks perform track-buffer read-ahead while the pseudodevice is delaying completion, accesses which are physically sequential will appear to have a faster than modelled transfer rate.

Our simulation results are shown in Figure 6. The improvements obtained by the manually modified applications increase steadily but insignificantly. This is unsurprising since their performance is limited by the available I/O bandwidth and their processing times are already only a small percentage of their execution times. The curves for the speculating applications are similar to those for the manually modified applications, although offset in Gnuld's case. For Agrep and XDataSlice, speculative execution already generates enough hints to keep the disks busy at all times.[5] For Gnuld, data dependencies, which are independent of processor speed, prevent speculative execution from using the additional cycles during I/O stalls to hint more read calls. For some applications, a more sophisticated design may be able to take advantage of these additional cycles. For example, it may prove useful to loosen our current definition of what it means for speculative execution to be on track. In general, however, applications dependent on recently read values may not be able to derive additional benefit from faster processors (unless they are rewritten to allow newly read data to affect future reads only after more intervening disk requests have been issued).

[5] Recall from the last section that the speculating Agrep was not able to generate enough hints to keep 10 disks busy on our current experimental platform. Under simulation, increasing the processor-to-disk speed ratio alleviated this problem so that, with a ratio of 3, the performance improvement of the speculating Agrep and the manually modified Agrep were 87% and 84%, respectively.

5 Future work

Having successfully demonstrated that speculative execution can be used to automate I/O hint generation, we are working on refining our design to better handle data-dependent applications like Gnuld. We discovered that even a simple, ad-hoc mechanism – disabling speculative execution for a brief time after some number of cancel requests have been issued – was sufficient to eliminate the performance penalty of performing speculative execution in Gnuld when the I/O system offered no parallelism. We are exploring more generic methods for limiting the number of erroneous hints generated, and for reducing the negative impact of erroneous hinting.

We are also investigating how speculative execution can be effectively employed in the range of possible multiprogramming/multithreaded scenarios. In particular, we are developing methods for evaluating the effectiveness of any particular speculation and for using this evaluation to decide what speculation, if any, should be scheduled and allowed to consume shared machine resources.

Multiprocessor environments offer another exciting possibility. One of the biggest challenges for proponents of multiprocessors is how they will enable non-parallelized applications to utilize the additional processing resources. By performing speculative execution in parallel with normal execution, disk-bound applications that cannot be automatically parallelized using compiler techniques may still be able to take advantage of the additional processing capabilities of a multiprocessor.

6 Related work

In Section 2, we discussed history-based prefetching, static approaches to automating prefetching, informing hints and the TIP prefetching and caching manager.

Mowry, Demke and Krieger's work [Mowry96] relies on static analysis, but also makes use of dynamic information provided by the operating system. Their approach applies to memory-mapped files, so that their hints affect virtual memory management as well as file cache management. Their use of hints differs from ours in that their compiler is responsible for placing hints based on a static decision of when prefetches should be issued, whereas we rely on TIP to manage the scheduling of prefetches.

Research presented by Franaszek, Robinson and Thomasian is close in spirit to our own [Franaszek92]. Through simulation, they demonstrated that pre-executing database transactions in order to prefetch data or pre-claim locks could significantly increase throughput because it reduced effective concurrency. However, their simulations assumed that pre-execution would always cause the correct data to be prefetched (or the correct locks to be claimed). Our approach differs from theirs primarily in two aspects. First, to reduce conflicts, they proposed that pre-execution of a transaction would run to completion before the transaction would re-execute with the intent to commit. In our system, pre-execution is overlapped with, and always secondary to, normal execution. Second, they explored pre-execution as a concurrency control technique for manual inclusion in the design and implementation of database systems. One of the essential properties of our work is the ability to automatically transform applications to use pre-execution.

The idea of adding software checks around load and store instructions was first brought to our attention by Lucco and Wahbe [Wahbe93]. They used these checks to perform software fault isolation, a fast alternative to hardware-enforced memory protection. Our checks are more complex and costly in order to implement software-enforced copy-on-write.

7 Conclusions

Disk-bound applications, increasingly common as faster computers and larger storage encourage users to manipulate more data, have their performance determined by storage rather than processor performance. While parallel storage systems are increasingly common, applications that exploit them well are not. Aggressive prefetching is a simple way to effectively utilize storage parallelism to reduce application latency, provided sufficiently detailed predictions of future accesses can be made sufficiently early.

This paper extends aggressive prefetching research with an automatic hint generation technique based on speculative pre-execution using mid-execution application state. Invoked only when the application is stalled waiting for I/O, speculative execution can add little or no observable overhead to the application. Provided that cycles are available in these time periods, speculative execution can discover future read accesses and issue hints to an aggressive prefetching system.

We have designed and implemented a binary modification tool that transforms Digital UNIX binaries to automatically perform speculative execution. Applied to a text search utility, a linker, and a 3-D visualization program, our system demonstrated 29% to 70% reductions in execution time with a four-disk array. A principle limitation of the current design is the lack of more effective automatic mechanisms for limiting the penalty of erroneous hinting due to data dependencies. The relatively large success of our currently unsophisticated design demonstrates that speculative execution is a promising new approach to aggressive I/O prefetching.

Acknowledgements

We thank David Nagle and Digital Equipment Corporation for providing the AlphaStation 500. We thank Paul Mazaitis for setting up the various hardware configurations, David Rochberg and Jim Zelenka for their assistance with TIP and Digital UNIX, and Robert O'Callahan for many invaluable discussions. We also thank John Hartman and the anonymous referees for their feedback on earlier drafts of this paper. TIP was developed by Hugo Patterson, and the SpecHint tool was inspired by a project with Steve Lucco to implement a software fault isolation tool for Digital UNIX.

References

[Baker91] Mary Baker, et. al. Measurements of a distributed file system. Proceedings of the 13th SOSP, October 1991.

[Cabrera91] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. Computing Systems 4(4), pp.405-436, Fall 1991.

[Cao94] P. Cao, E.W. Felton and K. Li. Implementation and performance of application-controlled file caching. Proceedings of the 1st OSDI. November, 1994.

[Cormen94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. TR PCS-TR94-243, Department of Computer Science, Dartmouth College, November 1994.

[Curewitz93] K.M. Curewitz, P. Krishnan and J.S. Vitter. Practical prefetching via data compression. Proceedings of the 1993 SIGMOD, May 1993.

[Feiertag71] R.J. Feiertag and E.I. Organisk. The Multics input/output system. Proceedings of the 3rd SOSP, 1971.

[Franaszek92] P.A. Franaszek, J.T. Robinson and A. Thomasian. Concurrency control for high contention environments. ACM TODS, V 17(2), pp. 304-345, June 1992.

[Gibson98] Garth Gibson, et. al. A cost-effective, high-bandwidth storage architecture. Proceedings of the 8th ASPLOS. October, 1998.

[Griffioen94] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. Proceedings of 1994 Summer USENIX, June 1994.

[Hartman94] John H. Hartman. The Zebra striped network file system. Doctoral thesis, UCB/CSD-95-867, December 1994.

[Kotz91] David Kotz and Carla Ellis. Practical prefetching techniques for parallel file systems. Proceedings of the 1st PDIS, December 1991.

[Kroeger96] T. Kroeger and D. Long. Predicting file system actions from prior events. Proceedings of 1996 Winter USENIX, January 1996.

[Lei97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. Proceedings of the 1996 Winter USENIX, January 1997.

[McKusick84] M.K. McKusick, et. al. A fast file system for UNIX. ACM TOCS, V 2(3), pp. 181-197, August 1984.

[Mowry96] Todd Mowry, Angela Demke and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. Proceedings of the 2nd OSDI, October 1996.

[Ousterhout85] J.K. Ousterhout, et. al. A trace-driven analysis of the UNIX 4.2 BSD file system. Proceedings of the 10th SOSP, December 1985.

[Paleczny95] M. Paleczny, K. Kennedy and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation, February 1995.

[Patterson88] David Patterson, Garth Gibson and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). Proceedings of the 1988 SIGMOD. June 1988.

[Patterson94] Hugo Patterson and Garth Gibson. Exposing I/O concurrency with informed prefetching. Proceedings of the 3rd PDIS. September, 1994.

[Patterson95] Hugo Patterson, et. al. Informed prefetching and caching. Proceedings of the 15th SOSP. December, 1995.

[Patterson97] Hugo Patterson. Informed prefetching and caching. Doctoral Thesis, CMU-CS-97-204, December 1997.

[Powell77] Michael L. Powell. The DEMOS file system. Proceedings of the 6th SOSP, November 1977.

[Rochberg97] David Rochberg and Garth Gibson. Prefetching over a network: Early experience with CTIP. ACM SIGMETRICS Performance Evaluation Review, V 25(3), pp. 29-36, December 1997.

[Thakur94] R. Thakur, R. Bordawekar and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. Workshop on I/O in Parallel Computer Systems, IPPS94, April 1994.

[Trivedi79] K.S. Trivedi. An analysis of prepaging. Computing, V 22(3), pp.191-210, 1979.

[Wahbe93] Robert Wahbe, et. al. Efficient software-based fault isolation. Proceedings of the 14th SOSP, December 1993.