Path-Based Target Prediction for File System Prefetching

Brian S. White Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{bsw9d,skadron}@cs.virginia.edu

Abstract

Prefetching is a well-known technique for mitigating the von Neumann bottleneck. In its most rudimentary form, prefetching simplifies to sequential lookahead. Unfortunately, large classes of applications exhibit file access patterns that are not amenable to sequential prefetching.

More general purpose approaches often use models to develop an appropriate prefetching strategy. Such models tend to be large, thus preventing a kernel implementation which would lead to user transparency and more efficient execution. This work applies the target cache approach of path-based branch target prediction to file system prefetching to combat these deficiencies. The feasibility and worth of such a design are evaluated against a number of parallel applications popular in the scientific community.

1 Introduction

Prefetching is a well-known technique for mitigating the von Neumann bottleneck. In its most rudimentary form, prefetching simplifies to sequential lookahead. Since most files tend to be accessed sequentially and in their entirety [1], this minimalist technique performs admirably in the common case.

Unfortunately, large classes of applications exhibit file access patterns that are not amenable to sequential prefetching. In particular, applications from the database and scientific communities often perform non-sequential, though regular, file access [17, 22]. `escat` [6] is an example of an application utilizing strided access patterns and will be examined below in some detail. Such behavior is characteristic of data sets which are partitioned amongst processors so that the partitions may be processed independently of one another, as in stencil algorithms [11].

A great deal of research has already addressed this I/O bottleneck. The mismatch between the traditional Unix-like I/O interface and the access patterns of parallel applications have lead to a re-evaluation of file system interfaces [10, 15, 16]. These interfaces present a higher level of abstraction than the traditional interface, thus facilitating the recognition of access patterns and allowing effective prefetching.

Unfortunately, the above approaches require code be rewritten to utilize the specialized interfaces. Such a requirement leads to portability concerns and additional burden for the application programmer. To address these issues, we may turn to automatic, transparent prefetching. Such techniques are completely transparent to the user and are embedded completely *within* the file system.

The latency disparity between processing and memory (and disk) access is evident at every level of the memory hierarchy. At the instruction level, the architect must be wary of branches which may destroy locality and lead to instruction cache misses (thus resulting in access to slower memory). Further, the processor often can not determine the branch target for several stages after instruction fetch. During this interim, instructions may be fetched along a particular path (likely the follow-through path). When the branch target is later determined, subsequent instructions may have to be flushed, as they are not on the program's control flow path.

There is an analogy between branches and file system requests. Waiting to determine the branch target until the last possible cycle may equate to a penalty in terms of wasted cycles while instructions are flushed. Similarly, demand I/O requests require that the application stall until the I/O has completed.

To avoid these penalties, computer architects have employed branch prediction and speculative execution. A branch predictor is accessed *before* the branch target is calculated. Instructions are then speculatively executed beginning at the predicted address. If the prediction is incorrect, the speculated instructions are later rolled back. However, if the prediction is correct, the processor has gained a few cycles of useful work (between the branch predictor access and the branch target calculation).

Branch predictors spanning a range of complexities have been proposed. Predictors for simple branches may be implemented as caches, addressed by branch address, and yielding 'Taken' or 'Not Taken' responses. Clearly, this simple binary choice does not map well to block prefetching. At any given point in a stream of I/O request, *any* block may be accessed. Indirect branches, such as those resulting from function pointer dereferencing, may exhibit such a range of targets. Predictors for these type of branches are often organized as target caches, addressed by both the branch address and some history of the last n branches. Instead of the binary response of more simple predictors, target caches store the actual target address of the branch.

This work applies the path-based indirect branch prediction, described above, to prefetching, in the hopes of preventing unnecessary blocking disk accesses. Instead of storing branch target addresses, the cache stores disk block addresses. Further, it is referenced by some representation of the last n disk request addresses. In this manner, the predictor automatically builds a model of the file access. Having established a model, the predictor approximates following a trace of accesses on that particular file. Such a predictor, implemented within the file system, maintains the desirable property of user transparency.

Section 2 highlights related work in the areas of prefetching and branch prediction. This section provides necessary background knowledge on branch prediction and its potential significance to prefetching. Section 3 presents the design of the path-based prefetch engine, which is evaluated through simulation in Section 4. Finally, Section 5 concludes.

2 Related Work

2.1 Prefetching

An optimal prefetching scheme has knowledge of all future disk accesses. Unfortunately, acquiring such a trace *a priori* is not possible. Nevertheless, applications may be able to provide hints regarding future accesses at runtime. For example, an application which reads a file in a strided fashion may *disclose* this knowledge to a file system in order that it make intelligent prefetching decisions. This concept forms the basis of Patterson's Transparent Informed Prefetching (TIP) [18].

Inaccurate prefetching may actually hurt system performance, since unneeded data may evict useful blocks from the cache. To this end, TIP provides a unified approach to caching and prefetching as advocated in [2]. Essentially, an application is instrumented with disclosures to the file system. The file system then evaluates the prefetching hint by considering the accuracy of past hints, the cost of ejecting blocks from the cache, and the opportunity cost of performing I/O [4].

Initial efforts required that the application be annotated with disclosures. This places a burden on the programmer and also requires source code availability. Automatic hint generation via optimizing compilers would alleviate the former restriction [19].

A recent approach overcomes both obstacles by instrumenting binaries [4]. Chang augments existing single-threaded applications with a *speculating thread*, acting as a companion to the original application thread. The speculating thread utilizes idle cycles, caused by the original thread's stalling for I/O, to speculatively execute past the blocking I/O. Upon encountering subsequent I/O, the speculating thread issues disclosure hints via the TIP mechanism. Some overhead is required to ensure that the two threads remained synchronized. Further, on a heavily loaded system, the low priority speculating thread may have little opportunity to execute. Future work will apply this technique to multi-threaded applications.

Other related work consists of online generation of access models. Vitter et al have applied data compression to prefetching, and proven its optimality in restricted cases [7, 23]. This work characterizes a sequence of page addresses as a stream of characters. The parse tree constructed from the sequence is used to prefetch pages and is represented by a probabilistic Finite State Machine or Markov source [23]. Given a cache size of k, this approach prefetches the k pages having the highest-probability edge assignments from the current node. This amount of prefetching (and its implications on the data cache) is alarming. It is unclear how such an obtrusive mechanism would thrive in the presence of multiple I/O threads, preceding at different rates. Each page request updates the tree and resets the current node to the root, where it is positioned by traversing the parse tree given the sequence substring.

Data structure size is a concern with this approach. The authors suggest using a 1st-order Markov predictor on a page sequence representing the last n accesses [7]. Such a predictor creates a parse tree based on the last n accesses, but considers only the most recent character in prefetching. The price to pay for a more realistic, bounded size implementation is a simpler (and likely less effective) approach.

Kroeger and Long have also employed the data compression approach to prefetching, albeit at a higher level [13]. Each event in their sequence represents a file open call. They perform whole-file caching and prefetch once the transition from a current node reaches a threshold value.

Following in the use of probabilistic models, Madhyastha calculates, via hidden Markov models, the probability distribution function of accesses, given an observed sequence of accesses [14]. The sequence is characterized at the granule of a cache block.

Kotz proposes a two-level dynamic predictor which recognizes simple patterns [12]. Local predictors attempt to classify patterns on a per-thread basis, while global predictors examine the global reference history resulting from the merger of local traces. For example, application threads may access a file in a strided manner, allowing the file system to prefetch at regular intervals. However, the agglomeration of such requests may represent sequential access. The local predictor assumes that all accesses are sequential and prefetches according, unless it detects a strided access.

2.2 Indirect Branch Prediction

Unconditional indirect branches have a potential for causing serious performance degradation in microprocessors. Unlike direct branches, a compiler or processor must determine not whether the branch will be taken, but what it's target will be. This latter point is trivial for direct branches, as they can have only one destination (other than the follow-through). Traditional branch prediction selects one of the two addresses based on knowledge of the current branch address and possibly local or global history.

Unfortunately, the target of an indirect branch can not be resolved via a simple binary decision. Instead, a target cache is employed [5]. Such a structure records a speculative target given (for example) a current branch address and a branch history. Chang, Hao, and Patt describe two means of addressing the cache: branch history and path history [5]. The branch history approach combines the branch address with global pattern history of the preceding conditional branches. Path history instead uses the target addresses of the last n branches.

Hash conflicts in target caches are more likely to be incorrect than in simpler BTBs, since they store actual target addresses. To prevent interference, target cache entries may be tagged (e.g. with the branch address). References to a hash entry are only valid if the query has a matching tag [5].

The information intended to be encapsulated by the index, a branch address and global history, is significantly larger than the index itself. [5, 8] propose intelligent hashing functions which attempt to retain the largest amount of information possible.

To avoid overly aggressive training, the target cache entries generally utilize a 2-bit hysteresis counter [8]. This counter is updated when the status of the prediction has been determined.

3 Path-Based Prefetch Engine

In examining the path-based prefetching approaches of the previous section, the community's concern with the potential size of the models is evident. Deriving a prefetch engine from a target cache has the pleasant side effect of introducing compactness and efficiency into the design. As a primary goal, this work hopes to leverage the frugal implementation of a hardware-based target cache to provide an unobtrusive footprint capable of being wired in (kernel) memory. Further, we strive for a user-transparent mechanism that prefetches effectively across a wide application domain.

As in the hardware approach, prefetching revolves around the central target cache data structure, which is organized as a collection of hash queues. Target cache entries are selected from a free list or LRU list before being added to a hash queue. Each entry has a target, a saturating (2-bit) counter, a tag field, and hash queue maintenance pointers (i.e. next and previous fields) and flags (i.e. `valid` and `free`). Unlike the hardware target cache, we need not resort to an associative cache to combat hash conflicts. Instead, the target cache supports fixed-depth hash chaining. This was done with the realization that a software-based implementation provides fewer execution time and complexity constraints, but also in an effort to avoid arbitrary-length traversals of kernel data structures.

For each open file, the kernel maintains a file structure on a per-process basis. This structure has been augmented for the purposes of implementing the path-based predictor. Namely, each file structure contains a path history 'register', a prefetch window, and pointers within these structures. Both windows wrap-around and are of fixed size (i.e. data which have not been processed are not overwritten). Of particular importance is the `lastn` pointer (where n is the number of entries hashed to access the target cache), which references the start of an n-length sequence in the path history register. This pointer is considered invalid if the path history register currently holds less than n elements. The path history register is twice as long as the number of entries needed to access the target cache (i.e. $2n$). In this way, the file system maintains the last n non-speculative disk accesses (referenced by `start`), followed by the last n speculative disk accesses (represented by `lastn`). These pointers are updated as appropriate on non-speculative and prefetching disk reads. In using the path history register to index the target cache, we use the most recent n entries. In many cases, these will correspond to speculative reads. While prefetching may be incorrect, entries are never added to the target cache until they have been verified correct.

The prefetch engine speculatively updates the path history register and `lastn` pointers on each prefetch. On each non-speculative read access, the file system compares the demanded block address against the head of the prefetch window. If there is a match, the pointers into the prefetch window are updated. Otherwise, the predictor has prefetched a block not (immediately) needed. The prefetch window is truncated and the path history register is 'rolled back'. That is, `lastn` is reset to `start`. This action may seem overly conservative. That is, a policy of penalizing the prefetch engine for failing to predict the correct *next* address may be too pessimistic. For example, if the prefetched block is needed within a short interval, the target cache has likely made a sound decision.

Having updated the prefetch window, and before updating the path history register, the file system addresses the target cache. At this point, it has determined the validity of the last prefetch. If the prediction was correct and the entry exists in the cache, the corresponding target cache entry's saturating counter is incremented. If the entry does not exist, it is created. However, if the prediction was incorrect, the target cache entry is decremented, and may be removed from the cache if its saturating counter has reached zero.

At this point, the file system requests the demanded disk block, updates the path history register, and invokes the prefetch engine. The prefetch routine executes until it has exhausted the number of empty entries in the prefetch window or it can no longer intelligently predict a next block address. At each iteration, the routine accesses the target cache using the (speculatively updated) path history register. If the target cache yields a valid target (i.e. a non-NULL entry whose saturating counter has reached its median value), that target is read *asynchronously*.

4 Experimental Methodology and Results

The path-based predictor was evaluated against a baseline file system via simulation. In an attempt to verify the design, a diverse class of I/O-bound applications was selected. The duration of their execution and the size of the data sets would have presented a challenge for anything but a simulation. Also, the number of parameters explored and number still to be explored suggest a trace-driven evaluation.

The next subsection details the selected application traces. Having developed an appreciation for their file access characteristics, we will examine their effect on tuning the predictor. Finally, we will evaluate the predictor against the baseline file system.

4.1 Application Traces

The following application traces were collected from the University of Illinois Pablo project [20] and the University of Maryland [21]. The original traces represented parallel applications executing on numerous application nodes. In all cases, a particular node was isolated and studied. In some cases, one node's I/O activity was focused predominantly around one file. In order to concentrate on the particular access pattern of interest, 'noise' disk activity was elided.

The resultant traces represent a large class of access patterns and applications. We describe them in the terminology of [22] as single-scan, strided-scan, multi-scan, triangle-scan, and oscillating-scan. These patterns are depicted in Figure 1.

4.1.1 `sar`

The Synthetic Aperture Radar (`sar`) technique uses aircraft- or satellite-mounted radar to bounce microwave signals off the ground to produce image data [20]. The application was originally traced on a 256-node Intel Paragon, using the `M_UNIX` iomode (i.e. each node has its own file pointer) of the PFS file system [20].

The nodes are organized into groups determined by the number of bands to be processed and the number of channels within the band. As is typical of parallel applications, one node reads initialization information and doles it out to other nodes via asynchronous communication.

A particular node within a group, the collector node, is responsible for reading a correction vector and transmitting this data to other members of the group. Polarization data is dealt out among the nodes in a group via the readers. Finally, the collector node writes output data to a file [20].

Despite its characterization as a single-scan algorithm [22], `sar` does not read the file in a strictly sequential manner.

4.1.2 `escat`

`escat` is a parallel implementation of the Schwinger Multichannel Election Scattering method [20]. The particular trace examined corresponds to trace `pSMCbp-A` from the Pablo web site, executing on a 256-node Paragon. In this version of the code, node 0 reads the data and broadcasts it to subsequent nodes. The data is partitioned among nodes, with each node accessing its particular sub-set via the PFS (shared file pointer) `M_ASYNC` mode. During the next phase, all nodes read via the (shared file pointer) `M_RECORD` mode and subsequently forward their data to node 0, where it is written [20]. This application reads data in a strided fashion.

4.1.3 `hartree`

The Hartree-Fock Code [20] is actually composed of three separate programs, only one of which is I/O-intensive. This program phase, `pscf` reads integrals and solves the corresponding equations. In so doing, it performs multiple iterations over the same file in a sequential fashion.

4.1.4 `lu`

`lu` performs LU decomposition on an out-of-core 8192 x 8192 double precision matrix using synchronous `read()`/`write()` requests [21]. As it iteratively sweeps the matrix, it performs a triangle scan on the file.

4.1.5 `titan`

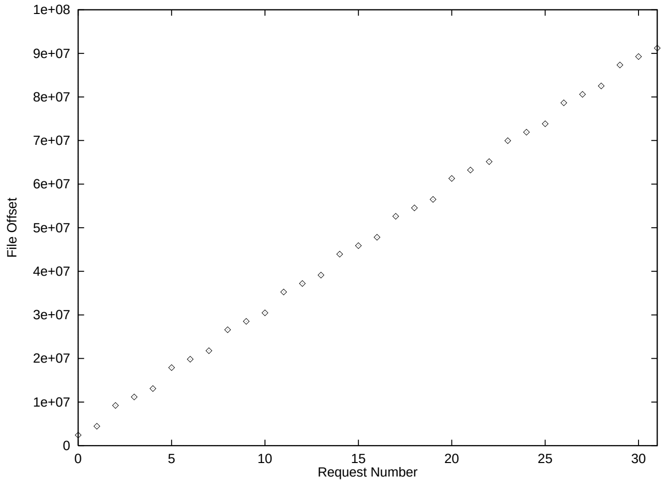
`titan` is a parallel scientific database for remote-sensing data. The application trace represents a series of queries run against the database. As is evident from the plot, the application exhibits the most randomized behavior of any examined trace.

4.2 Analysis of Parameter Instantiation

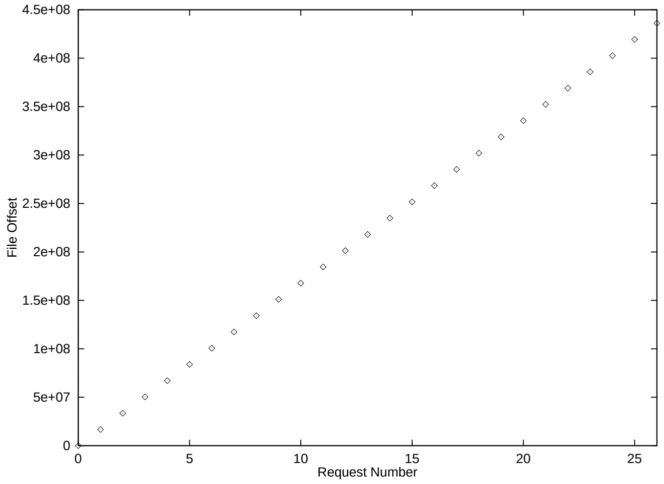
4.2.1 Second-Level Prediction

As described thus far, the path-based predictor was completely ineffectual.

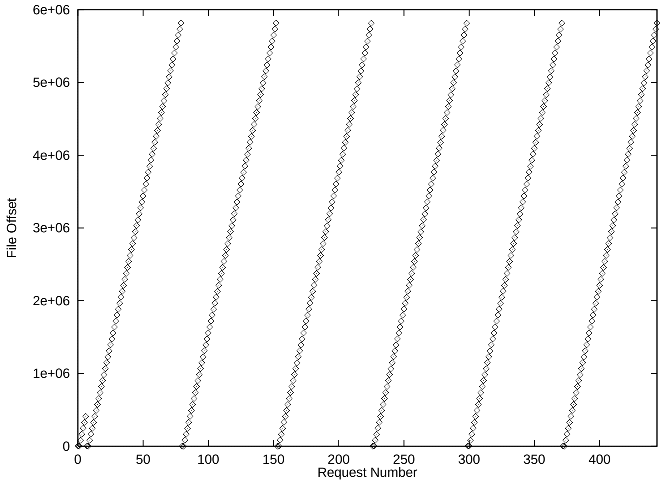
Two primary methods of encapsulating global history were considered. The first approach stores the branch target address of the last *n* accesses. `sar` and `escat` never reuse a target cache entry. Keeping in mind that a target cache entry must be verified correct *twice* before it will be used in prefetching, we would expect these two applications to gain nothing from the path-based predictor. Indeed, all applications missed on nearly *every* data cache access.



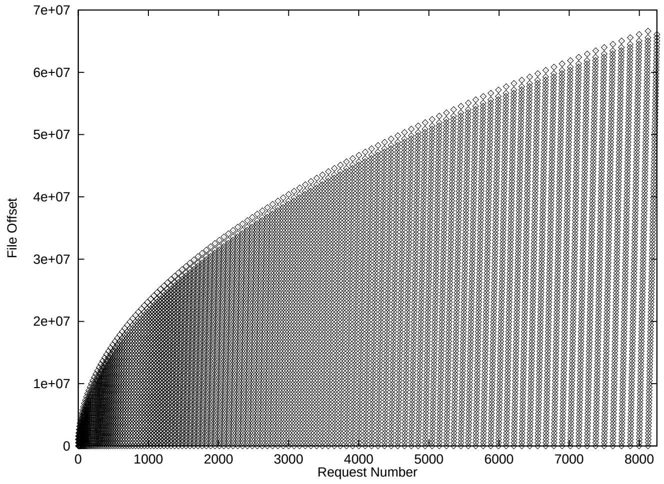
(a) single-scan(`sar`)



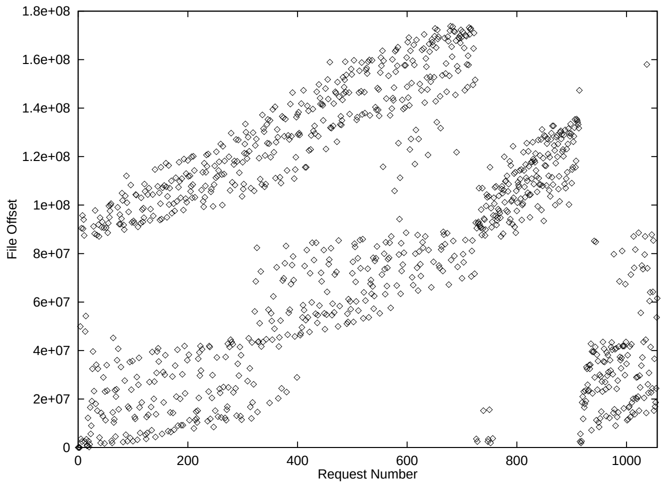
(b) strided-scan(`escat`)



(c) multi-scan(`hartree`)



(d) triangle-scan(`lu`)



(e) oscillating-scan(`titan`)

Figure 1: Application Traces

The second approach seeks to abstract away superficial differences within a file access sequence (e.g. disk block addresses) to uncover important similarities (e.g. a strided offset). Under this scheme, the offsets (or deltas) between contiguous blocks are stored in the path history register, instead of actual addresses. This was done in the hopes that the path history register would act as a template, automatically recognizing access patterns. For example, the path history register $< 1, 1, ..., 1 >$ would likely be a good indicator for the sequence $< 1, 2, 3, ... >$ *or* the sequence $< 88, 89, 90, ... >$. In this manner, it is possible to conserve target cache entries by consolidating the sequences $< 1, 2, 3, ... >$ *and* $< 88, 89, 90, ... >$ *and* a myriad of others into the one sequence $< 1, 1, ..., 1 >$.

Unfortunately, the potential now exists for a great deal of aliasing. To understand why, consider even the non-sequential file patterns above contain a good deal of sequential access. That is, the data points in Figure 1 represent calls to `read()` (or some parallel read operation). However, as these applications access large chunks of data, individual reads tend to be excessively large and span many disk blocks (sized `1024` in the simulation).

So, the sequential pattern occurs very frequently. If we hash only against this sequence, the number of cache collisions is intolerable. If we also hash against the current block (branch) address, we revert to the situation of many distinct cache entries representing the same semantic information.

There are two major problems brought to light above. First, for realistic target cache sizes, we will quickly exhaust all available resources by encoding redundant information. Second, the target cache has a steep learning curve. Consider a sequence of m sequential access, repeated for k iterations (somewhat analogous to `hartree`). If the current block address (or some other uniquifier) is hashed with the path history register, the target cache will not successfully predict the obvious sequential read-ahead until the third iteration through the loop (due to the hysteresis counter).

To overcome these obstacles, we turn to *cascaded predictors* [9]. A cascaded predictor is a form of two-level prediction, in which branch histories are first presented to a simple first-stage predictor. Only when this predictor is incorrect or fails to make a prediction, is the second-stage predictor invoked. Therefore, the presumably more valuable resources of the second-stage predictor are only used when they are absolutely required.

The first-stage predictor employed by the path-based predictor is simple indeed. This prediction routine simply looks for a path history register representing a straight sequential run. If it discovers such a run, it immediately predicts a sequential access. This prediction is *not* entered into the table; there is no need as the first-stage predictor can easily recreate the prediction. If the prediction later turns out to be incorrect, the corresponding path history register *will* be entered into the target cache (since it will necessarily suggest a non-sequential access).

Unlike the cascaded predictors of [9], the prefetch routine first accesses the target cache in making a prediction. If the target cache has no valid target, the prediction routine is invoked. Therefore, creating an entry for a sequential sequence whose target is non-sequential (as above), has the effect of blocking the first-stage predictor. Note, however, that we explicitly avoid entering histories (and associated speculative targets) representing sequential runs into the target cache (that is, sequential histories with a sequential target).

Performance improved markedly after making this simple change. It is assumed throughout the rest of the paper.

4.2.2 Addressing the Target Cache

This subsection examines indexing into the target cache. As mentioned above, the contents of the path history register itself are of extreme importance. Having introduced the first-level predictor, encoding the path history register with block offsets rather than block addresses is clearly more attractive. Further, this was borne out empirically.

However, as alluded to already, another value should likely be hashed along with the path history register to avoid hash collisions. Three candidates were examined: no additional hashing (NONE in Figure 2), storing the target block address corresponding to each offset in the path history register and hashing against each of these in turn (ALL), or hashing against only the most recent block address (CUR).

Since the hash function may interact with the path history length in an unforeseen manner, the path length was varied from 2 to 6. Further, for the time being, we do not want to be constrained by a limited cache size. The target cache size was set to a significantly large value so as to have no effect on the simulation. The results are displayed in Figure 2.

All applications experienced the lowest number of cache misses when using a path history register length of 3. This is consistent with the literature [8].

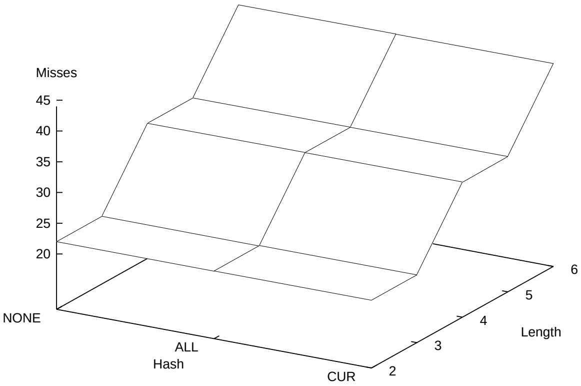
For all applications save `hartree`, hashing against the current block address yielded the best results. Subsequent simulations assumed these two parameters.

4.2.3 Target Cache Size

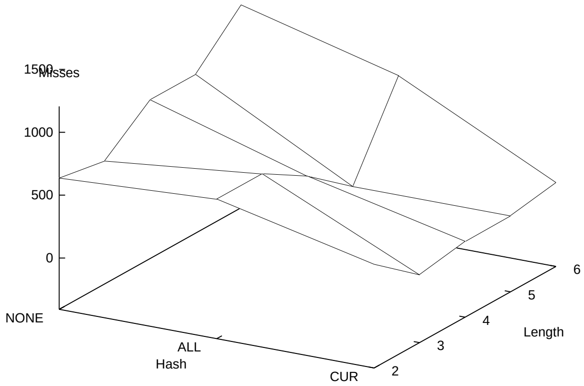
As stated, an important goal of this work is to minimize the size of an prefetching implementation. To this end, the effect of target cache size on cache misses was examined. 256-, 512-, 1024-, 2048-, and 4096-entry target caches were evaluated. There was a small increase in cache misses when using a 512-entry cache, and a similarly small increase when using a 256-entry cache. A 1024-entry cache is assumed for the rest of the experiments.

4.2.4 Tagged vs Tagless Target Cache

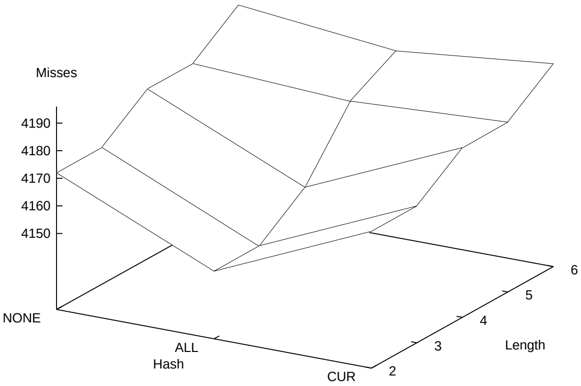
To further reduce the possibility of cache collisions, we consider using a tagged cache [5]. Once again, it appears that



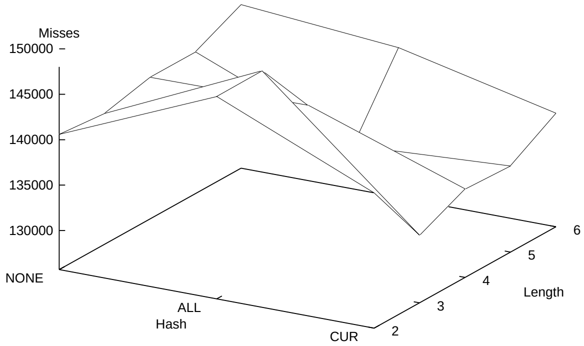
(a) `sar`



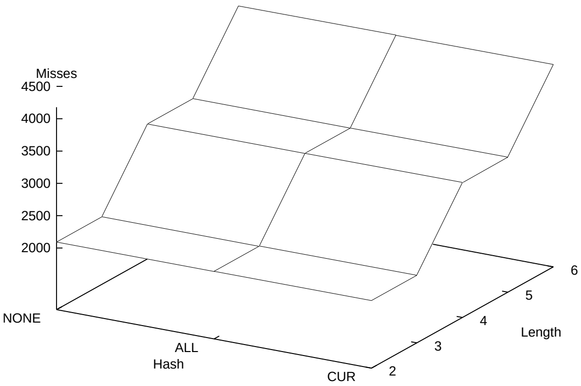
(b) `escat`



(c) `hartree`



(d) `lu`



(e) `titan`

Figure 2: Varying Hash Function and Path History Length

varying this parameter in isolation would ignore its potential interaction with other parameters, namely the currently posed method of avoiding collisions, hashing the current block address (CUR). Therefore, the effects on cache misses were studied while varying both of these parameters (Figure 3). NONE refers to a tagless cache. OFFSET refers to tagging the cache with the latest offset in the path history register. ADDR refers to tagging the cache with the previously seen block address.

The graphs for `escat`, `lu`, and `titan` have a surprisingly similar structure and support both tagging the cache and hashing with the latest block address. The sample space for `sar` is so small that it is not worth considering. Further, the differences in the plot of `hartree` are not as significant as they appear. In fact, the data point selected by the first three applications is only 30 cache misses from `hartree`'s minimum value. Therefore, the cache is tagged with the current block address and the path history register is hashed against this same address.

4.2.5 Prefetch Window Size

Varying the prefetch window size from 2 to 6 had no effect on the number of cache misses. This manifests one of the simplicities of the simulation. The simulator always assumes that there are sufficient I/O and data (cache block) resources to prefetch. Therefore, the prefetch engine always remains at least one access ahead of the blocking I/O calls (until an invalid prefetch is detected). Therefore, there is no reason that a prefetch window size any greater than one is necessary.

4.3 Baseline File System

In order to evaluate the merits of the design of the path-based predictor, a baseline file system simulator has been created as a point of comparison. This simulator attempts to mimic the Linux ext2 file system [3]. ext2 was selected because its prefetching appears to be fairly representative of what we would expect from a general-purpose file system. ext2 attempts sequential read-ahead of a block when it determines that recent accesses have been within a window of that block.

4.4 Results

The comparison is presented in Table 1. For all five of the applications surveyed, the path-based predictor performed better. We would expect the path-based predictor to perform significantly better on `escat` and `sar` since they exhibit very regular, non-sequential access patterns. The path-based predictor is able to recognize these patterns, while the baseline file system is not. True to our goal, the path-based predictor performs well in the presence of a diverse set of file access patterns. In fact, it performs better than the baseline on `hartree` which exhibits mostly sequential access.

The results reported are in terms of (disk block) cache misses. A cache miss was assigned only for synchronous I/O in which the requested disk block was not in the cache. Asynchronous I/O (including the prefetching of both file systems) did not contribute to cache misses.

Trace	Path-Based Predictor	Baseline
`sar`	22	730
`escat`	570	1053
`hartree`	4172	4202
`lu`	140107	213170
`titan`	2092	7400

Table 1: Blocking I/O Cache Misses

5 Conclusions and Future Work

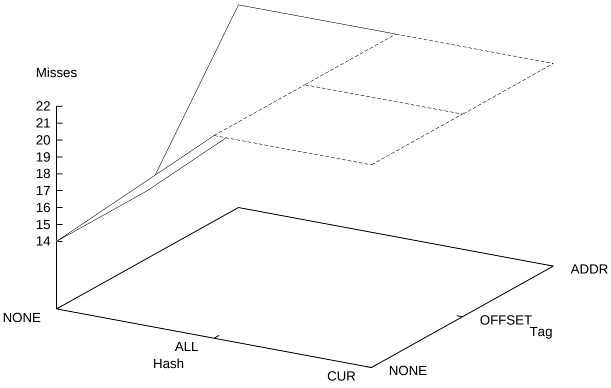
This work has shown the viability of using a small, path-based predictor in prefetching file blocks. Key to maintaining a minimal size is its use of offsets in the path history register (over target addresses) and incorporation of the first-level predictor. Further, a LRU replacement on the target cache entries ensures that infrequently used predictors do not pollute the cache.

This paper has shown through simulation the potential merits of the path-based predictive approach. However, several areas need to be addressed. In particular a more sophisticated simulator is needed to evaluate the predictor. The experiments enumerated above ignored the issue of block cache size. In general, a unified VM/buffer cache approach is required, as modern operating systems do not statically enforce the size of the cache. More importantly, the knowledge encapsulated in the target cache should be utilized in making page replacement decisions as in [2, 18]. Further, the prefetch engine should have a more sophisticated means of throttling itself based on current I/O and processor load and the potential cost of evicting a useful block. Such efforts would be greatly aided by a realistic disk model.

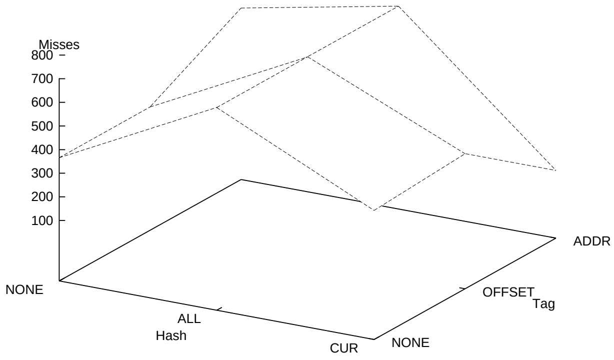
The means of evaluating the trace runs were overly simplistic. With a minimal effort it should be possible to determine a more meaningful metric than simply the number of cache misses. As stated, this raw number is misleading since the relative number of disk/memory addresses accessed by the target cache or the baseline is unknown. Further, there should be a penalty for accessing memory, or at least some indication of it reflected in the simulation results. For example, if one scheme has slightly fewer disk accesses, but many more memory accesses, we might consider it inferior.

It would be curious to consider the interactions between multiple processes and the target cache. For example, might we consider having a per-process target cache? Should there be a priority-based means of acquiring target cache entries in the face of competition for them?

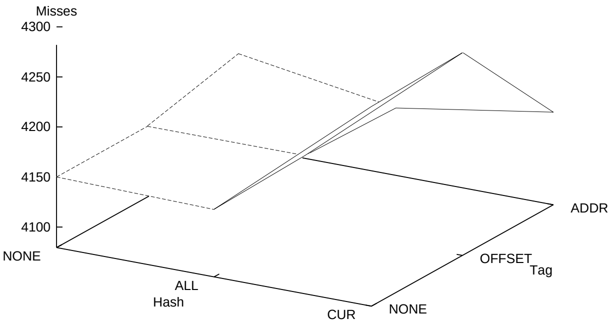
Finally, it would interesting to see how much the target cache can be shrunk and still be of value.



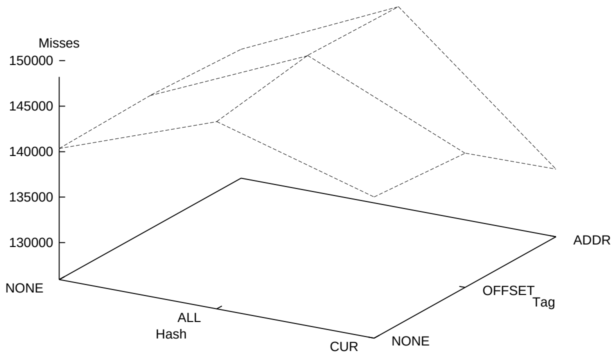
(a) `sar`



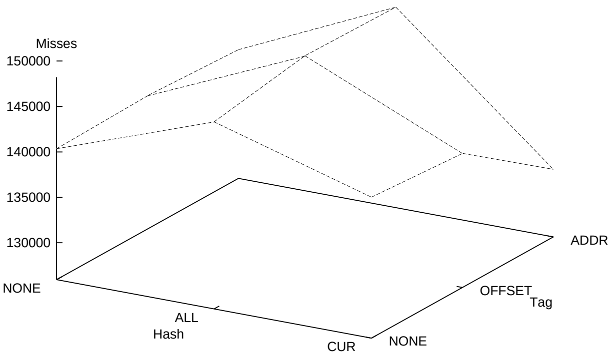
(b) `escat`



(c) `hartree`



(d) `lu`



(e) `titan`

Figure 3: Varying Hash Function and Tag

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 198–212, October 1991.

[2] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–196, Ottawa, Ontario, Canada, 1995.

[3] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. Linux Kernel Hacker's Guide Discussion Pages.

[4] Fay Chang and Garth A. Gibson. Automatic I/O generation through speculative execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation*. USENIX Association, February 1999.

[5] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *ISCA '97 Proceedings*, October 1997.

[6] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[7] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 1993.

[8] Karel Driesen and Urs Holzle. Accurate indirect branch prediction. In *ISCA '98 Proceedings*, July 1998.

[9] Karel Driesen and Urs Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *MICRO-31*, Dallas, TX, December 1998.

[10] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, October 1994. ACM Press.

[11] John F. Karpovich, Matthew Judd, W. Timothy Strayer, and Andrew S. Grimshaw. A parallel object-oriented framework for stencil algorithms. In *Proceedings of the Second IEEE International Symposium on High Performance Distributed Computing*, pages 34–41. IEEE Computer Society Press, July 1993.

[12] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[13] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[14] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, November 1997. ACM Press.

[15] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[16] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 9, pages 205–223. Kluwer Academic Publishers, 1996.

[17] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[18] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[19] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, April 1993.

[20] Daniel Reed. Pablo project experimental performance data. www-pablo.cs.uiuc.edu/Data/analyses.htm.

[21] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Maryland applications for measurement and benchmarking of i/o on parallel computers. www.cs.umd.edu/projects/hpsl/mambo/index.html.

[22] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Requirements of i/o systems for parallel machines: An application-driven study. Technical Report CS-TR-3802, University of Maryland, College Park, May 1997.

[23] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. In *Foundations of Computer Science*, pages 121–130, 1991.