ReplayCache: Exploiting Similarities for Predicting the Future

Ganesh C.N., Jaishankar Sundararaman, Ali R. Butt, and Godmar Back*
Dept. of Computer Science, Virginia Tech

1. INTRODUCTION

The access speed of mechanical disk devices continues to be orders of magnitude lower than that of random access memory. Consequently, successful caching and prefetching strategies are essential to achieving good I/O performance [1, 2]. Caching attempts to keep data in memory that will be accessed in the future, whereas prefetching initiates I/O requests for data an application is expected to access in the future. If the memory is not large enough to hold all the data accessed by the application, a cache replacement policy chooses which data to keep in the cache and which to replace.

Efficient caching and prefetching remain challenging issues because an application's future accesses are not usually known. Most caching strategies estimate the likelihood of future accesses based on some combination of recency, frequency, and patterns of past accesses. Recently, online prediction techniques [4] have been developed that use an application's current execution context (e.g., a snapshot of its call stack and current program counter location) to predict the pattern of accesses issued in that context. Although online prediction can recognize many patterns, it fails to capture complex I/O patterns, and often mispredicts patterns, leading to unnecessary prefetches or suboptimal replacement decisions [3, 4]. For example, if a repeated looping access pattern (best handled by keeping the accessed data in the cache) is misidentified as a sequential access (best handled by discarding data soon after use), overall application performance will be drastically reduced.

In this paper, we propose *ReplayCache*, an offline prediction framework that runs a workload in a *recording* mode, learning its access patterns and representing them in a way that makes it possible to execute a *similar* workload in *replay* mode. As an example, consider a typical scenario in which the program "make" is used to build a large C or C++ program out of different source files. An optimal caching and prefetching strategy would prefetch and cache all header files accessed by the C preprocessor. Temporary files, such as the `.i` files created by the preprocessor, or the intermediate `.s` assembly language files created by the C compiler, or the `.o` object files should be cached until they are accessed by subsequent stages, but can be purged from the cache soon after. If such a strategy could be learned from executing make on one project, it could benefit "make" processes with similar, though not identical execution sequences that operate on different sets of files. Using such a technique, users can record strategies for regularly run workloads — such as compilation, backups, compression, scripted multimedia or indexing workloads.

Our prediction technique relies on capturing the execution context that combines the current executable, current program counter, and call stack using a hash function, similar to PCC [4]. Our preliminary results indicate that these hashes are strong predictors from which optimal caching and prefetching strategies can be derived. These strategies can be compactly stored and applied with small runtime overhead. We have created a functioning prototype implementation of *ReplayCache* based on the FUSE [6] user-level filesystem infrastructure in Linux.

*{cnganesh, jaisunda, butta, gback}@cs.vt.edu

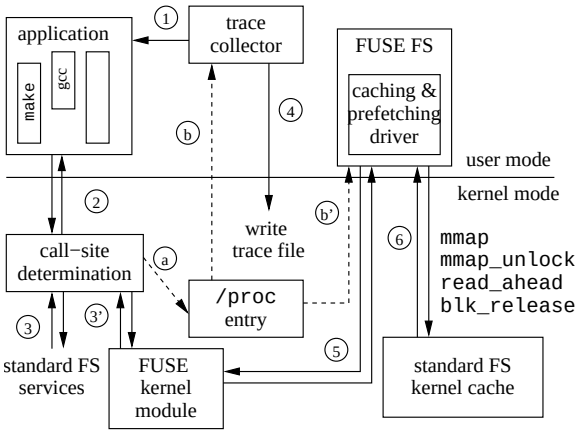

Figure 1: Key Components of ReplayCache

2. DESIGN

The primary components of the prototype we used to test the feasibility of our idea are shown in Figure 1. Most of the framework is implemented in user space, minimizing the amount of changes necessary to the underlying kernel.

Call-site determination and Tracing System: The call-site determination module is the only major component that is implemented in kernel space. In recording mode, the user-mode trace collector module first instantiates the application (Step 1 in the Figure). All `read()` and `write()` system calls from the application are intercepted (2), the appropriate signature (hash) for the current thread's call stack is determined and exported via a `/proc` entry (a). For each access, the module records the inode number to describe the file being accessed and the logical block number within the file. The system call is then handed over to the standard file system components for servicing (3). Simultaneously, the call-site signatures are read by the trace collector (b) and written to an application-specific trace file (4). Similarly, during the replay mode (2, 3', 5, 6), the call-site signatures are passed on to the prefetching and caching driver (a, b').

Pattern Predictor: The pattern predictor (not shown) post-processes and analyzes the traces recorded by the tracing system. It augments the traces by assigning virtual time values to each trace event and computes information such as reuse distance that can be used to predict patterns. This augmented trace file is then presented to one or more pattern predictors. We designed the pattern predictor interface to be pluggable, which provides the ability to choose from different recognition strategies.

Prefetching and Caching Driver: This component is implemented as a custom FUSE [6] module. During replay,

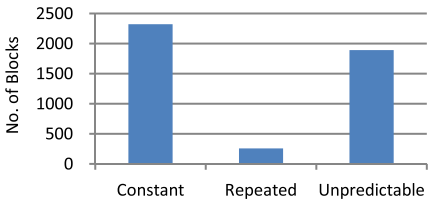

Figure 2: Predicting Reuse Number From Observed Call-site Signature

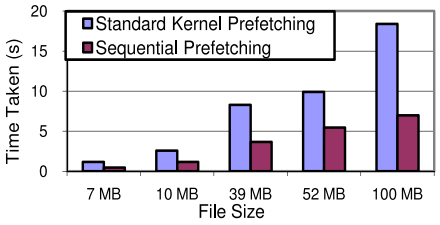

Figure 3: Synthetic Zig-Zag Load

it uses the strategy computed by the predictor and applies them to a particular workload. This driver performs the actual caching. It interacts with the kernel's page cache by memory mapping files to be cached into its address space. This mapping will create additional references to the file data in the page cache. In addition, by pinning the pages, we remove the kernel's ability to replace the pages, ensuring that they stay in physical memory. To replace a page in the cache when we predict that the data will not be accessed again, we unpin and unmap the file data first. Then, via a hook we added to the kernel's page cache, we request that the kernel move the block to the list of inactive pages (which are more likely candidates for eviction).

3. PATTERN PREDICTORS

Our goal is to find predictors that are strong, easy to compute, and which can be stored compactly and do not incur undue overhead when applied. In addition, these predictors must yield information that can lead to decisions that are beyond the capabilities of both conventional caching and prefetching algorithms and online prediction.

When making a caching decision, the first question to decide is whether a given block will be accessed repeatedly. Blocks that aren't accessed again should not be cached. Moreover, knowing the exact number of accesses to a block would allow targeted purging. For blocks that are reused, knowing the reuse distance can help rank cached data so that data with smaller reuse distance can be preferred if cached data must be replaced.

We investigated whether the call-site signature can predict which strategy to apply. The results of this investigation are shown in Figure 2 for a workload that traces `read()` calls for a run of "make" of the FUSE code itself. We weighed each PC signature by the amount of data that was accessed in that context. We partitioned the results in three groups. Group 1 (Constant) contains those signatures for which the number of accesses is a constant integer, allowing the strongest prediction of appropriate caching. Group 2 (Repeated) contains those signatures in which all blocks are accessed more than one time, but which do not fall in Group 1. These signatures hint that a block will be accessed again in the future, although it is impossible to say how often. Group 3 (Unpredictable) contains those signatures that contained both blocks that were not reused and blocks that were reused. For this group, we are unable to predict which caching strategy to use. Our results show that this simple strategy would have predicted an exact caching strategy for 51.9% of blocks, and would have provided reuse hints for an additional 5.7% of blocks.

Although state-of-the-art algorithms such as ARC [5] also attempt to differentiate between blocks that are accessed once from those that are accessed more than once, they can fail if the working set size is larger than the cache (+ history) size. If this happens, these algorithms may degenerate to a simple LRU scheme, which is known to make wrong decisions in such cases.

When making a prefetching decision, we must know if an access to a block predicts accesses to some or all of the other blocks in that file. For the workload considered above, we found—unsurprisingly—that all files are accessed in their entirety, making the call-site signature a perfect predictor by itself. By contrast, the default prefetching algorithm in the Linux kernel would only prefetch a file if either the file is small, or if a sequential access pattern is detected. Such mispredictions may incur considerable runtime cost. We used our prototype to measure the misprediction penalty for a synthetic "zig-zag" access pattern in which a file is accessed in its entirety, but with widely varying offsets between each access (as would be the case in a naive implementation of quicksort on a memory-mapped file). Figure 3 shows the results: whereas the Linux kernel fails to prefetch, simply engaging in a sequential prefetch— which we can predict using the call-site signature—could improve performance by 56%.

4. STATUS AND CONCLUSION

We believe that record and replay is a strategy that can improve the I/O performance of frequently recurring workloads, which often involve multiple applications. Neither recording nor replaying requires changes to the applications themselves. Our current prototype can record pattern and apply learned strategies. Our initial results indicate that even simple predictors have the potential for significant performance improvements. Our current research focuses on the development of more sophisticated predictors, on an evaluation of *ReplayCache*'s performance if the replayed workload exhibits larger differences from the recorded workload, and on a in-depth performance comparison with other, published caching strategies.

5. REFERENCES

[1] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE TOC*, 56(7):889–908, 2007.

[2] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.

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

[4] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proc. 6th USENIX OSDI*, 2004.

[5] Nimrod Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache In *Proc. USENIX FAST*, 2003.

[6] Miklos Szeredi. FUSE: Filesystem in user space. `http://fuse.sourceforge.net/`.